Modeling, Evaluating, and Improving the Performance of Supercomputer Scheduling

Thesis submitted for the degree of
“Doctor of Philosophy”

by

Dan Tsafrir

Submitted to the Senate of the Hebrew University
September 2006

This work was carried out under the supervision of Prof. Dror G. Feitelson

Dedication

To the loves of my life, Zohar and little Yoavi

Acknowledgments

First and foremost, I would like to thank my advisor Prof. Dror Feitelson. For his profound generosity. For his modest ways. For his quiet wisdom and brilliancy. For allowing me complete freedom, always being patient with me, and staying cool and humorous throughout. For the very many things that he taught me, and for being the role model that he is. I thank him from the bottom of my heart. It has truly been an honor.

Many thanks are due to Yoav Etsion (a.k.a. etsman), my roommate, extraordinarily talented partner, and friend. Aside from his significant contribution to my work, we have shared many of those day to day little things that, in the end, amount to the overall experience. We are brothers in arms. I was very fortunate to have such a companion.

I would like to thank the past and present members of the parallel systems lab: Ziv Balshai, Anat Batat, Donny Citron, Dudi Er-El, Yoav Etsion, Eitan Frachtenberg, Maayan Geffet, Avi Kavas, Uri Lublin, Avi Nissimov, Keren Ouaknine, Edi Shmueli, David Talby, and Yair Wiseman. I had a lot of fun during the many hours we have spent together. I especially cherish the (now mythological) basketball games we played using the lab's trashcan. Likewise, I would like to thank the system personnel, and notably Danny Braniss, Tomer Klainer, Eli Levy, Jorge Najenson, Ephraim Silverberg, and Chana Slutzkin. One cannot hope for a better system group (maybe the best on the planet). Their skills, expertise, and continuous willingness to provide immediate help were very much appreciated.

I am thankful to Prof. Jeff Rosenschein (under whom I served as a teacher assistant) for taking a chance on me. Teaching constituted a significant part of the PhD period and in this respect I feel I had the privilege to learn from the best, and that this has led to a very positive experience.

Finally, my warmest and deepest thanks go to my parents for a lifetime of love and support. There are no words to express how indebted and grateful I am to both of you. Thank you.

Abstract

The most popular scheduling policy for parallel systems is FCFS with backfilling (a.k.a. "EASY" scheduling), where short jobs are allowed to run ahead of their time provided they do not delay previously queued jobs (or at least the first queued job). This mandates users to provide estimates of how long jobs will run, and jobs that violate these estimates are killed so as not to violate subsequent commitments. The de-facto standard of evaluating the impact of inaccurate estimates on performance has been to use a "badness factor" $f \geq 0$, such that given a runtime r, the associated estimate is uniformly distributed in $[r, r \cdot (f+1)]$, or is simply $r \cdot (f+1)$. The underlying assumption was that bigger fs imply worse information.

Surprisingly, inaccurate estimates ($f > 0$) yield better performance than accurate ones ($f = 0$), a fact that has repeatedly produced statements like "inaccurate estimates actually improve performance" or "what the scheduler doesn't know won't hurt it", in many independent studies. This has promoted the perception that estimates are "unimportant". At the same time, other studies noted that real user estimates are inaccurate, and that system-generated predictions based on history can do better. But predictions were never incorporated into production schedulers, partially due the aforementioned perception that inaccuracy actually helps, partially because suggested predictors were too complex, and partially because underprediction is technically unacceptable, as users will not tolerate jobs being killed just because system predictions were too short. All attempts to solve the latter technicality yielded algorithms that are inappropriate for many supercomputing settings (e.g. using preemption, assuming all jobs are restartable, etcetera).

This work has four major contributions. **First**, we show that the "inaccuracy helps" common wisdom is merely an unwarranted artifact of the erroneous manner in which inaccurate estimates have been modeled, and that increased accuracy actually improves performance. Specifically, previously observed improvements turn out to be due to a "heel and toe" dynamics that, with $f > 0$, cause backfilling to approximate shortest-job first scheduling. We show that multiplying estimates by a factor translates to trading off fairness for performance, and that this reasoning works regardless of whether the values being multiplied are actual runtimes ("perfect estimates") or the flawed estimates that are supplied by users. We further show that the more accurate the values we multiply, the better the resulting performance. Thus, better estimates actually improve performance, and multiplying is in fact a scheduling *policy* that exercises the fairness/performance tradeoff. Regardless, multiplying is anything but representative of real inaccuracy, as outlined next.

Our **second** contribution is developing a more representative model of estimates that, from now on, will allow for a valid evaluation of the effect of inaccurate estimates. It is largely based on noting that human users repeatedly use the same "round" values (ten minutes, one hour etc.) and on the invariant that 90% of the jobs use the same 20 estimates. Importantly, the most popular estimate is typically the maximal allowed. As a result, the jobs associated with this estimate cannot be backfilled, and indeed, the more this value is used, the more EASY resembles plain FCFS. Thus, to artificially increase the inaccuracy one should e.g. associate more jobs with the maximum (a realistic manipulation), *not* multiply by a greater factor (a bogus boost of performance).

Our **third** contribution exploits the above understandings to devise a new scheduler that is able to automatically improve the quality of estimates and put this into productive use in the context of EASY, while preserving its attractive simple batch essence and refraining from any unacceptable assumptions. Specifically, the problem of underprediction is solved by divorcing kill-time from

the runtime prediction, and correcting predictions adaptively at runtime as needed, if they are proved wrong. The result is a surprisingly simple scheduler, which requires minimal deviations from current practices, and behaves exactly like EASY as far as users are concerned. Nevertheless, it achieves significant improvements in performance, predictability, and accuracy. Notably, this is based on a very simple runtime predictor that just averages the runtimes of the last two jobs by the same user; counterintuitively, our results indicate that using recent data is more important than saving and mining the history for similar jobs, as was done by previous work. For further performance enhancements, we propose to exploit the "heel and toe" understanding: explicitly using a shortest job *backfilled* first (SJBF) backfilling order. This directly leads to a performance improvements similar to those previously attributed to stunts like multiplying estimates. By still preserving FCFS as the basis, we maintain EASY's appeal and enjoy both worlds: a fair scheduler that nevertheless backfills effectively.

Finally, our **fourth** contribution has broader applicability, that transcends the supercomputing domain. All of the above results are based on the standard methodology of modeling and simulating real activity logs of production systems, which is routinely practiced in system-related research. The overwhelmingly accepted assumption underlying this methodology is that such real workloads are representative and reliable. We show, however, that real workloads may also contain anomalies that make them non-representative and unreliable. This is a special case of multi-class workloads, where one class is the "real" workload which we wish to use in the evaluation, and the other class contaminates the log with "bogus" data. We provide several examples of this situation, including an anomaly we call "workload flurries": surges of activity with a repetitive nature, caused by a single user, that dominate the workload for a relatively short period. Using a workload with such anomalies in effect emphasizes rare and unique events (e.g. occurring for a few days out of two years of logged data), and risks optimizing the design decision for the anomalous workload at the expense of the normal workload. Thus, we claim that such anomalies should be removed from the workload before it is used in evaluations, and that ignoring them is actually an unjustifiable approach.

Contents

Dedication iii
Acknowledgments iv
Abstract v

Contents **vii**
List of Publications x
Preface xii

1 Introduction **1**
1.1 Background 3
1.1.1 Job Scheduling 3
1.1.2 Backfilling 5
1.2 Motivation 9
1.2.1 The Unresolved Mystery of Inaccurate Estimates 9
1.2.2 The Failure to Model the “Badness” of User Estimates 11
1.2.3 The Failure to Improve the Quality of Estimates for Backfilling 13
1.2.4 The Problematic Nature of Raw Workload Data 16
1.3 Preview of Results 18
1.3.1 Solving the Mystery of Why Inaccuracy May Help 18
1.3.2 Modeling Estimates 21
1.3.3 Incorporating System-Generated Predictions in Backfill Schedulers 24
1.3.4 Workload Flurries and Sanitization 28

2 Methodology **31**
2.1 The Trace Files 31
2.2 The Simulator 32
2.3 Simulating EASY Backfilling 33
2.4 Performance Metrics 34
2.5 Artificially Varying the Load 35

3 Solving the Mystery of Why Increased Inaccuracy May Help **36**
3.1 Introduction 36
3.2 Performance as a Function of Badness 37
3.3 Backfilling as a Function of Badness 39
3.4 The Heel-and-Toe Dynamics 40
3.5 Countering the SJFness of Heel-and-Toe 42

3.6 The Role of Burstiness 45
3.7 Unfairness as a Function of Badness 47
3.8 Making the Model More Realistic 48
3.9 Practical Implications 51
3.10 Non-FCFS Backfilling 55
3.11 Conclusions 57

4 Backfilling With System-Generated Predictions 58
4.1 Introduction 58
4.2 Incorporating Predictions into Backfilling Schedulers 60
4.2.1 Separating the Dual Roles of Estimates 61
4.2.2 Prediction Correction 62
4.2.3 Shortest Job Backfilled First (SJBF) 64
4.2.4 Varying the Load 65
4.2.5 Optimizations Summary 66
4.3 Predictability 66
4.4 Relationship With Other Algorithms 68
4.5 Does Better Accuracy Imply Better Performance/Predictability? 70
4.6 Tuning Parameters 72
4.7 Conclusions 76

5 Modeling User Runtime Estimates 77
5.1 Introduction 77
5.2 Existing Estimate Models and Their Shortcomings 79
5.3 Methodology and Roadmap 81
5.4 Input, Output, and Availability 82
5.5 Trace Files Manipulation 82
5.6 Mass Disparity of Estimates 83
5.7 Number of Estimates 85
5.8 Time Values of Estimates 86
5.9 Popularity of Estimates 88
5.10 Mapping Time to Popularity 90
5.10.1 Mapping of Tail Estimates 90
5.10.2 Determining Head Times 91
5.10.3 Mapping of Head Estimates 93
5.10.4 Embedding User-Supplied Estimates 95
5.11 Overview of the Model 96
5.11.1 About the Complexity 97
5.12 Validating the Model 97
5.12.1 Validating the Distribution 97
5.12.2 Assigning Estimates to Jobs 99
5.12.3 Validating Performance Results 99
5.12.4 Repetitiveness is Missing 101
5.13 Conclusions 102

6 Workload Flurries and Data Sanitization 103
6.1 Introduction . . . 103
6.2 A Case Study of Instability . . . 105
6.2.1 Example of a Butterfly Effect . . . 105
6.2.2 The Role of a Flurry in Causing the Effect . . . 106
6.2.3 Explaining the Sensitivity . . . 108
6.3 The Phenomenon of Workload Flurries . . . 109
6.4 Impact of Flurries on System Evaluation . . . 111
6.5 On Why the Removal of Flurries is the Right Thing to Do . . . 112
6.5.1 How About Removing Entire Days? . . . 113
6.5.2 Standard Alternatives are More Aggressive . . . 113
6.5.3 How About Removing Just the Anomalous Part of the Days? . . . 114
6.5.4 How About Not Removing Anything and Separate Averages Instead? . . . 115
6.5.5 How About Not Separating the Averages and Shake the Input Instead? . . . 117
6.6 Impact of Flurries on Modeling . . . 117
6.7 Generalizing . . . 118
6.8 Conclusions . . . 121

7 Discussion and Conclusions 122
7.1 Resolving the Misconception of Inaccurate Estimates . . . 122
7.2 Accurately Modeling User Runtime Estimates . . . 124
7.3 Leveraging System-Generated Predictions for Backfilling . . . 125
7.4 Cleaning Workloads From Flurries and Other Anomalies . . . 127

Bibliography 130

List of Publications

1. ***"Secretly monopolizing the CPU without superuser privileges"***
   Dan Tsafrir, Yoav Etsion, Dror G. Feitelson
   USENIX Security Symposium
   Aug 2007, Boston, Massachusetts (to appear)

2. ***"The context-switch overhead inflicted by hardware interrupts (and the enigma of do-nothing loops)***
   Dan Tsafrir
   *ACM Workshop on Experimental Computer Science (**ExpCS**)*
   Jun 2007, San-Diego, California (to appear)

3. ***"Backfilling using system-generated predictions rather than user runtime estimates"***
   Dan Tsafrir, Yoav Etsion, Dror G. Feitelson
   *IEEE Transactions on Parallel and Distributed Systems (**TPDS**)*
   Jun 2007, pages 789–803, volume 18, number 6

4. ***"Fine grained kernel logging with KLogger: experience and insights"***
   Yoav Etsion, Dan Tsafrir, Scott Kirkpatrick, Dror G. Feitelson
   ACM EuroSys
   Mar 2007, Lisbon, Portugal

5. ***"Process prioritization using output production: scheduling for multimedia"***
   Yoav Etsion, Dan Tsafrir, Dror G. Feitelson
   *ACM Transactions on Multimedia Computing, Communications and Applications (**TOMCCAP**)*
   Nov 2006, pages 318–342, volume 2, number 4

6. ***"The dynamics of backfilling: solving the mystery of why increased inaccuracy may help"***
   Dan Tsafrir, Dror G. Feitelson
   *IEEE International Symposium on Workload Characterization (**IISWC**)*
   Oct 2006, San Jose, California, pages 131–141

7. ***"Instability in parallel job scheduling simulation: the role of workload flurries"***
   Dan Tsafrir, Dror G. Feitelson
   *IEEE International Parallel and Distributed Processing Symposium (**IPDPS**)*
   Apr 2006, Rhodes Island, Greece, page 10

8. ***"Workload sanitation for performance evaluation"***
   Dror G. Feitelson, Dan Tsafrir
   *IEEE International Symposium on Performance Analysis of Systems and Software (**ISPASS**)*
   Mar 2006, Austin, Texas, pages 221–230

9. ***"System noise, OS clock ticks, and fine-grained parallel applications"***
   Dan Tsafrir, Yoav Etsion, Dror G. Feitelson, Scott Kirkpatrick
   *ACM International Conference on Supercomputing (**ICS**)*
   Jun 2005, Cambridge, Massachusetts, pages 303–312

10. ***"Modeling user runtime estimates"***
    Dan Tsafrir, Yoav Etsion, Dror G. Feitelson

*Workshop on Job Scheduling Strategies for Parallel Processing (**JSSPP**)*
Jun 2005, Cambridge, Massachusetts, pages 1–35, Lecture Notes in Computer Science, volume 3834

11. ***"Desktop scheduling: how can we know what the user wants?"***
Yoav Etsion, Dan Tsafrir, Dror G. Feitelson
*ACM International Workshop on Network and Operating Systems Support for Digital Audio and Video (**NOSSDAV**)*
Jun 2004, Kinsale, Ireland, pages 110–115

12. ***"Effects of clock resolution on the scheduling of interactive and soft real-time processes"***
Yoav Etsion, Dan Tsafrir, Dror G. Feitelson
*ACM International Conf. on Measurement and Modeling of Computer Systems (**SIGMETRICS**)*
Jun 2003, San Diego, California, pages 172–183

13. ***"Barrier synchronization on a loaded SMP using two-phase waiting algorithms"***
Dan Tsafrir, Dror G. Feitelson
*IEEE International Parallel and Distributed Processing Symposium (**IPDPS**)*
Apr 2002, Fort Lauderdale, Florida, page 80

14. ***"Reducing performance evaluation sensitivity and variability by input shaking"***
Dan Tsafrir, Keren Ouaknine, Dror G. Feitelson
Technical Report 2007-24, School of Computer Science and Engineering, the Hebrew University
May 2007

15. ***"Session-based, estimation-less, and information-less runtime prediction algorithms for parallel and grid job scheduling"***
David Talby, Dan Tsafrir, Zviki Goldberg, Dror G. Feitelson
Technical Report 2006-77, School of Computer Science and Engineering, the Hebrew University
Aug 2006

16. ***"A short survey of commercial cluster batch schedulers"***
Yoav Etsion, Dan Tsafrir
Technical Report 2005-13, School of Computer Science and Engineering, the Hebrew University
May 2005

17. ***"General purpose timing: the failure of periodic timers"***
Dan Tsafrir, Yoav Etsion, Dror G. Feitelson
Technical Report 2005-6, School of Computer Science and Engineering, the Hebrew University
Feb 2005

18. ***"Workload flurries"***
Dan Tsafrir, Dror G. Feitelson
Technical Report 2003-85, School of Computer Science and Engineering, the Hebrew University
Nov 2003

Preface

The research projects I have been involved in are related to systems of varying scale, ranging from small personal desktops, through modest SMPs, to large scale supercomputers, and deal with several different aspects of the studied systems (see publication list above). Aggregating all the projects into a single document would have violated the space constraints of a PhD dissertation, and therefore we have chosen to focus on only a few closely related papers that can enjoy a higher-level collective presentation. The chosen theme concentrates on the larger systems, and in particular, on how to improve the performance of schedulers of supercomputers.

Chapter 1

Introduction

The most commonly used scheduling algorithm for supercomputers is FCFS (First-Come First-Served) with backfilling, which requires users to provide runtime estimates of how long their jobs will run. The estimates are used by the scheduler to better “pack” the jobs and therefore one would naturally assume that accurate values would yield better packing and hence improve the overall utilization and turnaround times. However, a decade-old mystery that repeatedly surfaces suggests otherwise: it turns out many independent researchers have found time and again that increasingly inaccurate estimates actually improve performance. Consequently, statements in the spirit of “with respect to backfilling, what the scheduler doesn’t know won’t hurt it” [34] have become widespread.

This dissertation makes four major contributions. **First**, we resolve the estimates mystery and show it is merely the result of an unrealistic side effect of the manner by which increasingly inaccurate estimates are artificially manufactured. Therefore, our **second** step is developing an alternative model that allows for a true evaluation. The results are in explicit disagreement with past findings, and prove better accuracy does in fact translate to superior performance. This conclusion motivates searching for a way to improve the quality of the (notoriously poor) user estimates, used by backfill schedulers. All previous attempts to accomplish this task have failed, due to technical limitations inherent to backfilling. Our **third** contribution is overcoming these difficulties, while leveraging the mechanics underlying the now-resolved mystery of why performance appeared to improve. The suggested scheduling scheme (which utilizes system-generated runtime predictions instead of user estimates) is remarkably similar to, and enjoys all the benefits of, plain FCFS with backfilling. Nevertheless, it significantly improves accuracy, predictability, and performance (which is up to doubled).

Our findings are based on the modeling, analysis, and simulation of real workload logs recorded on real production systems. This methodology is standard and heavily used by numerous studies, under the assumption that such logs are reliable and representative. Our **fourth** and final major contribution is discovering this assumption is actually often false, as logs might also contain anomalies that make them non-representative and unreliable. One important recurring anomaly is what we call “workload flurries” (very rare surges of activity with a repetitive nature, caused by a single user). We show that basing an analysis on workloads including such anomalies can lead to bogus evaluation results and eventually to bad system designs. Consequently, in contrast to the common practice, we advocate that production logs be “sanitized” before being used, by deleting these anomalies (similarly to the removal of outliers in statistical analysis).

#	*topic*	*motivation*		*preview of results*		*chapter*		*referred paper*	
		sect.	*page*	*sect.*	*page*	*#*	*page*	*ref.*	*venue*
1	Solving the inaccuracy mystery	1.2.1	9	1.3.1	18	3	36	[159]	IEEE Int'l Symp. on Workload Characterization (**IISWC'06**), to appear
2	Modeling user runtime estimates	1.2.2	11	1.3.2	21	5	77	[157]	LNCS Workshop on Job Scheduling Strategies for Parallel Processing (**JSSPP'05**)
3	Backfilling with system predictions	1.2.3	13	1.3.3	24	4	58	[156]	IEEE Transactions on Parallel & Distributed Systems (**TPDS'07**), to appear
4	Flurries and data sanitization	1.2.4	16	1.3.4	28	6	103	[160]	IEEE Int'l Parallel & Distributed Processing Symp. (**IPDPS'06**)
								[54]	IEEE Int'l Symp. on Performance Analysis of Systems & Software (**ISPASS'06**)

Table 1.1: *The four topics covered by this dissertation. Each topic is introduced in this chapter by two subsections: "motivation" and "preview of results" (2-3 pages per subsections). Later, each topic is covered by a dedicated chapter, which is largely based on the associated paper(s).*

Dissertation Roadmap The layout of this dissertation revolves around the four aforementioned topics, as shown in Tab. 1.1: each topic is associated with a separate chapter, which in turn is associated with (and largely based on) one or two of the papers we have chosen to include in this dissertation, as mentioned above. Other than these four chapters, the dissertation includes another three: this one, which introduces the work, the next one, which presents our methodology, and the last one, which contains our concluding remarks.

Introduction Chapter Roadmap Similarly to the higher level presentation approach, this chapter's structure is also largely based on the above division to four topics. It is composed of three section: background (1.1), motivation (1.2), and preview-of-results (1.3). The background is divided into two parts: it first provides a general survey on job scheduling (Subsection 1.1.1), and then zooms in on the issue of backfilling, which is the most popular scheduling scheme and the immediate setting for this work (Subsection 1.1.2). The motivation section is divided into four subsections, one per topic, each discusses in detail the various aspects and related work, which motivated us to investigate that topic. Finally, the preview-of-results section is also subdivided into four, such that each subsection presents our key findings in relation to the associated topic.

We note that a considerable effort has been put into making this chapter self-contained: the "motivation" subsections fully introduce the four topics, and a real effort was made such that the associated "preview-of-results" subsections would clearly explain the more important bottom-lines and findings. (In contrast, the remaining chapters largely build on the material presented here.)

1.1 Background

This chapter first presents a general overview on the topic of job scheduling (Sec. 1.1.1), and then focuses on surveying backfill schemes (the immediate context of this work), and in particular, on the EASY scheduler.

1.1.1 Job Scheduling

Supercomputers A parallel computer is “a collection of processing elements that communicate and cooperate to solve large problems fast” [3]. The main motivation for developing and using such computers is that whatever the performance of a single processor at a given time, higher performance can, in principle, be achieved by utilizing many such processors. Parallel machines are often referred to as “supercomputers”, if the number of processors composing them is relatively high. Installations with tens to hundreds of processors are commonplace nowadays, and the “top-500 list” (which lists the 500 most powerful supercomputers in the world) is dominated by machines with thousands or more processors and is lead by the 131,072 processors BlueGene/L [29]. It has been established that the combined power of the top-500 almost doubles every year [43] and that the combined number of processors is doubled every three years [42]. Thus, it is reasonable to expect similar trends in lower-end supercomputers that are in common use, ever increasing the supercomputing power.

Jobs Supercomputers are nowadays used to execute diverse tasks including weather forecasting and climate research (e.g. about global warming), molecular modeling (e.g. of structures and properties of chemical compounds, biological macromolecules, polymers, crystals etc), various physical simulations (e.g. airplanes in wind tunnels, detonation of nuclear weapons, nuclear fusion), cryptanalysis, data mining, and more. Parallel applications, called “jobs”, are usually composed of a number of independent sequential processes that execute simultaneously, each on a different processor. While they run, the processes communicate and exchange information from time to time, in an effort to complete the task as soon as possible. Other types of parallel applications that do not require communication between the computing parties also exist (e.g. workpile applications like database transactions), but are not the focus of this work. Rather, we are interested in parallel jobs in their “traditional” or “scientific” sense, where processes actually cooperate in order to solve the problem.

Workload A supercomputer is a scarce and relatively expensive resource. At the same time there are many potential users that can benefit from having access to supercomputing capabilities. In an attempt to both accommodate users’ collective needs as well as to maximize the utilization of what is essentially an expensive machine, modern systems allow multiple users (typically hundreds [110]) to simultaneously use the same supercomputer. As a consequence, the workload of a supercomputer generally consists of a sequence of jobs that are submitted for execution by several users at arbitrary times. This scenario is often referred to as being “on-line”, namely, that the submission times of jobs (also called their “arrival times”) are a-priori unknown [51]. Thus, each job is characterized first and foremost by its arrival time.

Partitioning To support the execution of multiple jobs, most contemporary production systems employ what is known as “space slicing” or “space sharing” [67, 164, 73, 44, 2, 108], meaning that each job is allocated a “partition” of the machine (a subset of its processors) for its exclusive use.

After allocation, the parallel job runs to completion, in batch mode, without being interrupted or preempted. The exact size of the partition is not enforced by the system. Rather, it is set according to the user's explicit request, a policy called "variable partitioning". Thus, a second defining attribute of a job (in addition to its arrival time) is its "size", namely, the number of processors it requires in order to run. This value is provided by the user upon the job submittal.

Note that there exist other partitioning strategies that, while an immense research effort has been invested in them, are far less common. For example, even though the mainstream practice is to allow only "rigid" jobs (for which users specify a single possible size), a recent survey [23] revealed that 98% of the jobs are in fact "moldable" (fit more than one partition size). Such information can certainly be used by the supercomputer to improve its utilization [16, 120, 167, 168, 121, 127, 134, 112, 23] and this ability was even incorporated in some research platforms [45], however, we are unaware of any production system that makes use of it.

Rigid and moldable jobs are suitable for "static partitioning", where the chosen size applies throughout the entire execution of the job and is never changed. In contrast, "evolving" jobs [116, 66] and "malleable" jobs [100, 14, 52, 80] must be supported by environments that employ "dynamic partitioning", which allows for a change in the size of the jobs during runtime [111, 32, 65, 102, 140, 107, 25]. (The difference between the two type is that for evolving jobs, changes are application-initiated, whereas for malleable jobs the decision to change the number of processors is made by an external job scheduler.) One variant of dynamic partitioning called "equipartitioning" (strives for equal partition sizes for all jobs) was repeatedly shown to consistently produce good results [103, 18, 28, 27, 113, 119]. Nevertheless, dynamic partitioning calls for a radical change in the programming model, and we speculate that most users will not even consider structuring their programs in a way that complies with this paradigm. To the best of our knowledge, the sole implementation of dynamic partitioning on a production machine was done in research context using the CM-5 Connection Machine [13].

Not only the space of the machine can be partitioned, but also its time. Environments that support "time sharing" or "time slicing" allow sharing of processors between jobs by means of context switching [109, 128, 104, 88, 119]. Under this title, the "gang scheduling" policy, which insures all the processes of a job are executed simultaneously (by preempting and rescheduling them at the exact same time) has drawn a lot of attention [49, 7, 50, 69, 59, 126, 170]. In contrast to the various dynamic partitioning alternatives that turned out to have mostly theoretical value, gang scheduling (under the static partitioning discipline) has proved to be more practical: Firstly, it has been implemented as an optional part of the scheduling algorithm within a range of real systems (SGI's IRIX [8], Intel's Paragon [165], IBM's LoadLeveler resource manager [82], and Cray's XD1 [153]). Secondly, it has been utilized in experimental systems that were actually deployed in several sites (LLNL's BBN-Butterfly [63] and Cray-T3D [78]). And lastely, there is at least one known, efficient, and widely deployed gang scheduling implementation — on the CM-5 Connection Machine [95, 108].

Despite this relative success, time slicing inevitably involves various nontrivial difficulties that are absent from strict space-slicing/static-partitioning alternatives. A notable example is how to avoid memory contention in the face of having more than one job simultaneously using the memory [165, 9]. Such difficulties are probably the reason why production systems predominantly favor the space-slicing/static-partitioning, as will be further discussed next.

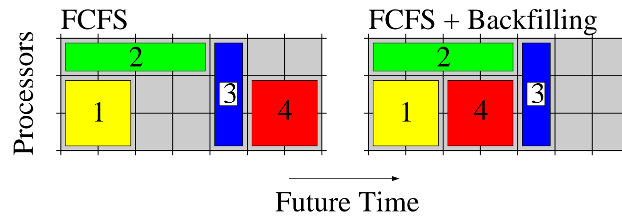


Figure 1.1: *A space/time Gantt chart displaying a FCFS schedule without (left) and with (right) backfilling. Each rectangle represents a job, such that the rectangle's width and height are the job's runtime and size, respectively. The job numbers indicate arrival order (not arrival time). Obviously, backfilling reduces fragmentation and improves the utilization. Note, however, that it would have been impossible to backfill job 4 had its length been more than 2 time units, as the reservation for job 3 would have been violated.*

1.1.2 Backfilling

EASY Backfilling The default algorithms used by current job schedulers for parallel supercomputers are all rather simple and similar to each other [37], employing a straightforward version of variable partitioning. (Recall that this means space-slicing with static-partitioning, where users specify the number of processors required by their jobs upon submittal.) In essence, schedulers select jobs for execution in first-come first-served (FCFS) order, and run each job to completion, in batch mode. The problem with this simplistic approach is that it causes significant fragmentation, as jobs with arbitrary sizes/arrivals do not pack perfectly. Specifically, if the first queued job requires many processors, it may have to wait a long time until enough are freed. During this time, processors stand idle as they accumulate, despite the fact there may very well be enough of them to accommodate the requirements of other, smaller, waiting jobs.

To solve the problem, most schedulers therefore employ the following algorithm. Whenever the system status changes (job arrivals or terminations), the scheduler scans the queue of waiting jobs in order of arrival (FCFS) and starts the traversed jobs if enough processors are available. Upon reaching the first queued job that cannot be started immediately, the scheduler makes a *reservation* on its behalf for the earliest future-time at which enough free processors would accumulate to allow it to run. This time is also called the *shadow time*. The scheduler then continues to scan the queue for smaller jobs (require fewer processors) that have been waiting less, but can be started immediately without interfering with the reservation. In other words, a job is started out of FCFS order only if it terminates before the shadow time and therefore does not delay the first queued job, or if it uses extra processes that would not be needed by the first queued job. The action of selecting smaller jobs for execution before their time provided they do not violate the reservation constraint is called *backfilling*, and is illustrated in Fig. 1.1 (see detailed description in Section 2.3).

This approach was initially developed for the IBM SP1 supercomputer installed at the Argonne National Laboratory as part of *EASY* (Extensible Argonne Scheduling sYstem), which was the first backfilling scheduler [98].[1] The term "EASY" later became a synonym for FCFS with backfilling against a reservation associated with the first queued job. (Other backfill variants are described below.) While the basic concept is extremely simple, a comprehensive study involving 5 supercomputers over a period of 11 years has shown that consistent figures of 40–60% average utilization have gone up to around 70%, once backfilling was introduced [79]. Further, in terms

[1] Backfilling has later been integrated with the IBM LoadLeveler scheduler for the SP2 system [60], and has been supported ever since.

of performance, backfilling was shown to be a close second to more sophisticated algorithms that involve preemption (time slicing), migration, and dynamic partitioning [19, 170].

User Runtime Estimates The down side of backfilling is that it requires the scheduler to know in advance how long each job will run. This is needed for two reasons:

1. to compute the shadow time for the longest-waiting job (e.g. in the example given in Fig. 1.1, we need to know the runtimes of job 1 and job 2 to determine when their processors will be freed in favor of job 3), and
2. to know if smaller jobs positioned beyond the head of the wait-queue are short enough to be backfilled (we need to make sure backfilling job 4 will not delay job 3, namely, that job 4 will terminate before the shadow time of job 3).

Therefore, EASY required users to provide a runtime estimate for all submitted jobs [98], and the practice continues to this day. Importantly, jobs that exceed their estimates are killed, so as not to violate subsequent commitments (the reservation). This strict policy has the additional benefit that it supplies an inherent and clear motivation for users to provide high quality estimates , as short enough values increase the chances for backfilling, but too-short values will get jobs prematurely killed.[2]

Popularity of EASY The burden placed on users to provide estimates has not been detrimental. Rather, the combination of simplicity, effectiveness, and FCFS semantics (often perceived as most fair [123]) has made EASY a very attractive and a very popular job scheduling strategy. Nowadays, virtually all major commercial and open-source production schedulers support EASY backfilling [37], including

- IBM's LoadLeveler [60, 82],
- Cluster Resources' commercial Moab [118] and open-source Maui [75] (which is probably the most popular scheduler used within the academia),
- Platforms' LSF (Load Sharing Facility) [172, 24],
- Altair's PBS (Portable Batch System) [68] in its two flavors: commercial PBS-Pro [33] and open-source OpenPBS [10], and
- Sun's GridEngine [61, 106]

The default configuration of all these schedulers, except PBS, is either EASY or plain FCFS (with FCFS, however, the schedulers' behavior becomes EASY if backfilling is nevertheless enabled). The CTO of Cluster Resources has estimated that 90-95% of Maui/Moab installations do not change their default (EASY) settings [74]. Being the exception that implies the rule, the PBS variants use Shortest-Job First (SJF) as their basic default policy. However, even with PBS, when a job is "starved" (a situation defined by PBS to occur if the job is waiting for 24 hours or more) then the scheduling reverts to EASY until this job is started. As a testament for its immense popularity, a survey about the top 50 machines within the top-500 list revealed that, out of the 25 machines for which relevant information was available, 15 (= 60%) were operating with backfilling enabled [36].

[2]Indeed, the administrator guide of e.g. LSF clearly states that "Since jobs with a shorter run limit have more chance of being scheduled as backfill jobs, users who specify appropriate run limits will be rewarded by improved turnaround time." [24]

Variations on Backfilling Despite the simplicity of the concept, backfilling has nevertheless been the focus of dozens of research papers attempting to evaluate and improve the basic idea.[3] We do not list them all here, but rather, cite many of them (and more) when appropriate, within their respective contexts later on. The remainder of this section only briefly mentions some of the various tunable knobs of backfilling algorithms.

One tunable parameter is the **number of reservations**. As mentioned above, in EASY, only the first queued job receives a reservation. Thus, backfilling may cause delays in the execution of other waiting jobs which are not the first and therefore do not get a reservation [47]. The obvious alternative is to allocate reservation to all the jobs. This approach has been named "conservative backfilling" as opposed to the "aggressive" approach taken by EASY [108]. However, it has been shown that delaying other jobs is rarely a problem, and that conservative backfilling tends to achieve reduced performance in comparison to the aggressive alternative. The MAUI scheduler includes a parameter that allows system administrators to set up the number of reservations [75]. It has been suggested that allocating up to four reservations is a good compromise [15].

A second parameter is the **looseness of reservations**. For example, an intriguing suggestion is a "selective reservation" strategy depending on the extent different jobs have been delayed by previous backfilling decisions. If some job is delayed by too much, a reservation is made for this job [141]. This is somewhat similar to the "flexible backfilling" strategy, in which backfilling is allowed to violate the reservation(s) up to a certain slack [150, 166]. (Setting the slack in the latter strategy to be the threshold used for allocating selective reservations in the former strategy, is more or less equivalent.)

A third parameter is the **order of queued jobs**. EASY, as well as many other system designs, use FCFS order [98]. A general alternative is to prioritize jobs in a certain way, and select jobs for scheduling (including as candidates of backfilling) according to this priority order. For example, flexible backfilling combines three types of priories: an administrative priority to favor certain users or projects, a user priority used to differentiate between the jobs of the same user, and a scheduler priority used to guarantee that no job is starved [150]. The Maui scheduler has a priority function that includes even more components [75]. Another approach is to prioritize based on various job characteristics. In particular, a set of criteria related to the current queueing time and expected resource consumption of jobs has been proposed, which generalizes the well-known SJF algorithm for improved performance [174, 115] as well as combines it with fairness notions [19, 15]. The queuing order and the timing of reservations can also be determined by economic models [35] or various quality of service assurances [72].

A fourth parameter (related to the previous one) is the **partitioning of reservations**. The processors of a machine can be partitioned into several disjoint sets (free processors can dynamically move around between them based on current needs). Each set is associated with its own wait-queue and reservation. Lawson and Smirni divided the machine such that different sets serve different jobs classes, characterized by their estimated runtime (e.g. short, medium, and long) [90, 92]. A backfilling candidate is chosen in a round-robin fashion, each time from a different set, and must respect all reservations. By separating short from long jobs, this multiple queue policy reduces

[3] For example, searching the ACM digital library for papers with "backfill" appearing in their title or abstract results in a (far from complete) list of more than 30 papers, most are directly dealing with the subject. The query "(backfill OR backfilling) AND parallel AND scheduler" in Google's Scholar retrieves more than 400 documents, of which the overwhelming majority are related to job scheduling. The initial paper about EASY [98] is listed by Scholar as cited 158 times.

the likelihood that a short job is overly delayed in the queue behind a very long job, and therefore improves average performance metrics.

A fifth parameter is the **adaptiveness of backfilling**. An adaptive backfill scheduler continuously simulates the execution of recently submitted jobs under various scheduling disciplines, compares the hypothetical resulting performance, and periodically switches the scheduling algorithm to be the one that scored the highest. In the face of different workload conditions, this adaptiveness has the effect of both improving and stabilizing the observed performance results [144, 149].

A sixth parameter is the amount of **lookahead into the queue**. Most backfilling algorithms consider the queued jobs one at a time when trying to backfill them, which often leads to loss of resources to fragmentation. The alternative is to consider the whole queue at once, and try to find the set of jobs that together maximize the utilization while at the same time respecting the allocated reservation(s). This appears to be a NP-hard problem, but due to the fact machine sizes are relatively small, this can be done in polynomial time (in the complexity of the machine size) using dynamic programming, leading to optimal packing [132, 131].

A seventh and final parameter is related to **speculative backfilling**, where the scheduler is allowed to exploit gaps in the schedule for backfilling, even if the backfilled job interferes with the reservation. By doing so, the scheduler speculates that the backfilled job would terminate sooner than its estimate suggests, and in any case before the shadow time. Successful speculations obviously improve performance and utilization, and have no negative side effects. But unsuccessful speculations must somehow be dealt with. Unfortunately, all previously suggested solution resulted in a scheduling algorithm that lies outside the attractive variable partitioning domain: The simplest alternative is to kill the offending backfilled job and restart it later on [90]. A similar idea is to employ "short test runs", during which jobs either manage to terminate, or are reinserted to the wait-queue with a tighter estimate deduced from the test [115, 15]. Unfortunately, both ideas assume jobs are restartable, which is often not the case. A possible workaround is to employ preemption (time sharing), such that instead of killing and restarting the job, it is suspended and kept in memory, only to be resumed later on from the point in which it was stopped [139]. However, if preemption comes into play, it may be preferable to instead combine backfilling with a full fledged gang scheduler altogether [169]. This combination has even been further extended by adding migration capabilities [85, 170]. A recent study suggested preemption is actually redundant if migration or dynamic partitioning are available. The idea is to reduce the backfilled job's processor allocation by folding it over itself. This frees most of its processors, and limits the performance degradation to the offending job [162].

Finally, a new direction in job scheduling research is to try and **minimize the electric power demand**, which is rapidly becoming a problem in the context of supercomputing [129]. It has been suggested to integrate the concept of scheduling and power management within EASY [91]. The proposed scheduler continuously monitors load in the system and selectively puts certain nodes in "sleep mode" (makes them unavailable for execution), after estimating the effect of fewer nodes on the projected job slowdown. Using online simulation, the system adaptively selects the minimal number of processors that are required to meet certain negotiated service level agreements.

Backfilling and Grid Computing We note in passing that the backfilling doctrine naturally fits into grid settings were certain assurances are often needed regarding start times of waiting jobs. (In fact, the "reservation" term is heavily used in gird context.) For example, this is needed for co-allocation, where a job is simultaneously scheduled to run on multiple remote cites [83, 137, 96].

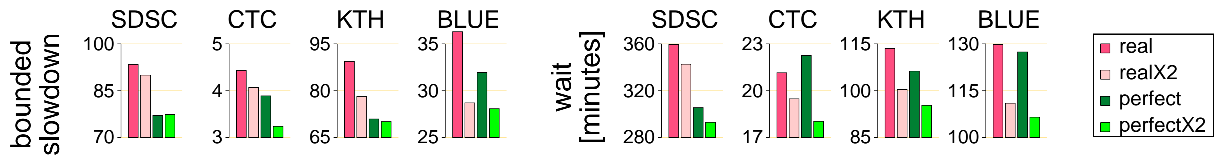


Figure 1.2: *The average wait time and slowdown of all jobs obtained when simulating four different workloads, both with real user estimates ("real") and after replacing them with actual runtimes ("perfect"; f=0). In both cases making estimates less accurate by doubling ("X2") tends to help. (See the next chapter for a detail description of both the simulator and the workloads.)*

1.2 Motivation

This work focuses on backfill scheduling algorithms and in particular, due to its immense popularity, on EASY as their representative. Recall that backfilling requires users to provide runtime estimates, used by the scheduler to better pack the jobs. Attempts to assess the impact of inaccurate estimate on performance have yielded a very surprising result: it appears that increased inaccuracy improves the performance, as described next.

1.2.1 The Unresolved Mystery of Inaccurate Estimates

Modeling Inaccuracy In 1998, Feitelson and Mu'alem-Weil proposed the "f-model" in order to study the sensitivity of backfilling to the quality of estimates [47]. Given a job J with runtime r, the model postulates that its estimate is chosen at random from a uniform distribution in the range $[r, (f+1)\cdot r]$, where $f \geq 0$ is a predetermined constant.[4] They termed f the "badness factor" because estimates become increasingly inaccurate as f grows, with $f = 0$ indicating completely accurate estimates. The f-model has been used when simulating workloads that lacked estimates data [169, 56, 58], but much more importantly, the model and its variants have been extensively used to study the impact of inaccurate estimates on backfilling algorithms [146, 47, 174, 108, 15, 142, 170, 122, 34, 64]. One simple variant of interest is the "deterministic f-model", in which there is no random component and estimates are simply set to be $(f+1)\cdot r$, that is, a direct multiple of the associated runtime and some factor [174, 15, 34].[5]

The Inaccuracy Mystery A very surprising result repeatedly reported by the aforementioned papers was that, in terms of performance, inaccurate estimates are usually preferable over accurate ones. This is illustrated in Fig. 1.2. Evidently, performance improves when deliberately making estimates less accurate by doubling them. This is true both when doubling perfectly accurate estimates and when doubling the original (inaccurate) user estimates.[6]

While there is a wide agreement that making estimates less accurate by multiplying them with some factor is usually beneficial, the effect of the chosen f is less obvious. This is illustrated

[4] f is nonnegative because a job is killed by the system if it tries to run beyond its estimate, so the estimate is never smaller than the runtime.

[5] The first known use of the deterministic model was by Suzuoka et al. [146] in 1995 (the same year in which the first paper about EASY was published [98]), which utilized artificial estimates that are 50% bigger than real runtimes ($f = \frac{1}{2}$ in badness terminology) to evaluate the impact of inaccuracy.

[6] In 1999, Zotkin and Keleher conjectured that the improvement obtained when multiplying *perfect* estimates by some factor (as was reported in 1998 by Feitelson and Mu'alem-Weil [47]), might also be obtained if multiplying *real* user estimates [174]. This was later verified to be true by Mu'alem and Feitelson in 2001 [108].

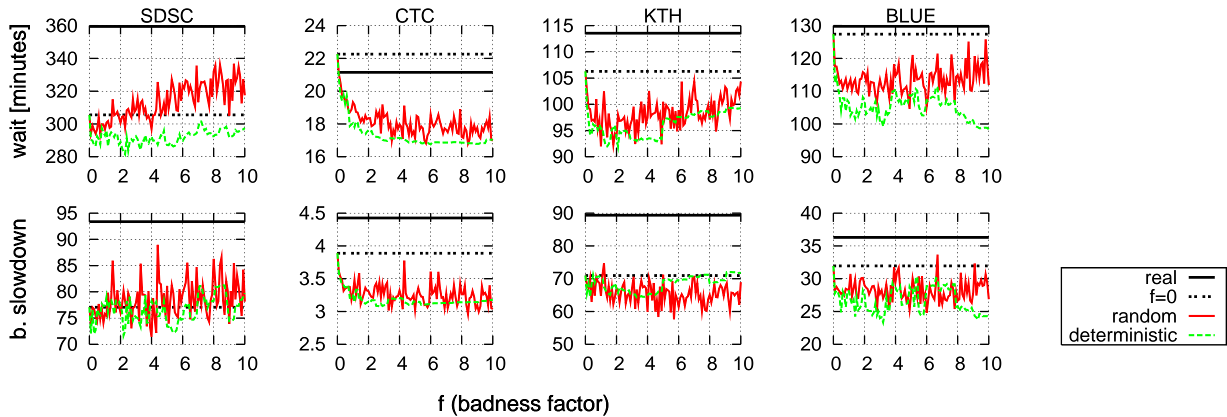


Figure 1.3: *Performance as a function of f for the random and the deterministic f-model ("real" corresponds to real user estimates). Excluding SDSC, most results associated with positive f values are better (smaller) than the performance associated with f=0.*

in Fig. 1.3. Faced with (usually a small subset of) such results, researchers claimed that the improvement in performance is largely "insensitive" to f [174, 169, 34, 64]. Further, England et al. suggested a new "robustness" metric for evaluating the performance of computer systems and claimed (in one case-study demonstrating the usefulness of their metric) that [34]:

ROBUSTNESS CLAIM

"Our results support those of a previous work and also indicate that backfilling is robust to inaccurate run time estimates in general. It seems that, with respect to backfilling, what the scheduler doesn't know won't hurt it."

The Failure to Explain the Mystery The fact nonzero badness ($f > 0$) usually improves performance was unanimously explained by what we call the *"holes argument"* [47, 174, 108, 15, 142], as elegantly articulated by Chiang et al. [15]:

HOLES ARGUMENT

"We note that for large f (or when multiplying [real] estimates by two), jobs with long runtimes can have very large runtime overestimation, which leaves larger 'holes' for backfilling shorter jobs. As a result, average slowdown and wait may be lower"

At the same time, the observed "insensitivity" of performance to the exact badness value for $f > 0$, was explained by what we call the *"balance argument"* [174, 169, 170, 64], as articulated by Zhang et al. [169]:

BALANCE ARGUMENT

"We can understand why backfilling is not that sensitive to the estimated execution time by the following reasoning. On average, overestimation impacts both the jobs that are running and the jobs that are waiting. The scheduler computes a later finish time for the running jobs, creating larger holes in the schedule. The larger holes can then be used to accommodate waiting jobs that have overestimated execution times. The probability of finding a backfilling candidate effectively does not change with the overestimation."

For example, doubling the lengths of all the jobs in Fig. 1.1 only means the X-axis is scaled by a factor of two, but doesn't change anything regarding the backfilling decision: indeed, after doubling, job 4 looks twice as long in the eyes of the scheduler, but the same applies to the 2-time-units-hole opened by job 2, so job 4 can still backfill.

While both arguments seemingly make sense, one obvious problem with them is that they are contradictory: If the balance-argument is correct, then there is no benefit in opening those "larger holes" as suggested by the holes-argument, because backfilling candidates would become proportionally larger and cancel the effect. On the other hand, the "holes argument" implies a performance improvement that is proportional to f, in contrast to the balance-argument rationale. Regardless of the contradiction, both arguments fail to explain the results shown in Fig. 1.3, for example the noisiness of BLUE (performance is actually quite sensitive to f), or the opposite trends observed in SDSC/wait vs. CTC/wait (CTC/wait supports the holes-argument while SDSC/wait contradicts it; both contradict the balance-argument).

1.2.2 The Failure to Model the "Badness" of User Estimates

The Role of a Model The purpose of a model is to truthfully reflect reality, thereby allowing a valid performance evaluation methodology. A successful model makes it possible to artificially generate a *representative* workload, similar to the activity typically experienced by the relevant systems. The output of a model is then used as the input (benchmark) that drives the evaluated system, which can be an actual existing system or a simulated one. Simulations are especially important for evaluating system designs, for which the chosen configuration is yet to be determined (often based on simulation results). The main advantage of using a model in this context is its flexibility: changing the configuration of a planned system is usually as easy as changing the value of a parameter (e.g. memory size, number of processors, etc). This methodology allows, for example, to decide upon the most cost-productive system configuration, before purchasing it.

There is another potentially significant benefit to developing a model that manages to successfully characterize the inherent nature of a representative workload: the understandings and insights it exposes can often lead to the design of an improved system, which is better suited to efficiently handle this type of a workload.

The Failure of the f-Model According to Fig. 1.3, the popular f-model fails to achieve its objective, as it yields unrealistically improved performance results that are consistently better than those obtained when real user estimates are utilized (the only exception is very small f values in CTC/wait). Recall that the purpose of a model is not to paint a bright picture of reality, but rather, an accurate one. Surprisingly, this deficiency has usually been brushed aside. And so, in contrast to the other key parameters of parallel workloads (jobs' runtimes, interarrival times, number of processors) that received *a lot* of attention in terms of realistic modeling [46, 30, 77, 17, 20, 170, 99, 105], the dominant estimate model has been the f-model [146, 47, 174, 169, 108, 15, 142, 170, 122, 34, 64], or simply using actual runtimes instead of estimates (f=0) [84, 145, 163, 39, 133]. We conjecture that this can be partially attributed to the perception that estimates are unimportant because "inaccuracy improves performance" and "what the scheduler doesn't know won't hurt it". (Paradoxically, these statements are largely based on research that utilizes the f-model itself.) But the results associated with real estimates (Fig. 1.3) clearly demonstrate that reality is more complex.

The Failure of the Φ-Model Recall that backfill schedulers kill underestimated jobs to insure reservations are respected, and that this policy creates a clear motivation for users to supply accurate estimates. This is true because (1) jobs would have a better chance to backfill if their estimates are tight, but (2) would be killed if they are too short. The popularity of EASY and the fact it is used in production systems owned by organizations that agree to share their scheduling logs, have made it possible to evaluate whether this incentive manages to actually deliver high quality estimates. The consistent conclusion was negative, namely, user estimates turned out to be rather poor [47, 17, 108, 20, 38, 93, 157]. This is demonstrated in Fig. 1.4 using data from the four different installations used earlier. The graphs are histograms of the estimation *accuracy*: what percentage of the "requested time" (as embodied in estimates) was actually used. The promising peak at 100% regrettably reflects jobs that reached their allocated time and were then killed by the system according to the backfilling rules. The hump near zero reflect very short jobs (less than 90 seconds) that failed on startup. The rest of the jobs, that actually ran successfully, have a rather flat uniform-like histogram, meaning that for such jobs, any level of accuracy is almost equally likely.

Noticing the failure of the f-model, Mu'alem and Feitelson attempted to develop a more successful model by recreating the histograms shown in Fig. 1.4 [108]. The histogram's flat portion implies that $r/e = u$, i.e., that the ratio of the actual runtime r to the estimate e can be modeled as a uniformly distributed random variable in the range $u \in (0, 1]$. By changing sides we get $e = r/u$, so given a runtime r we can generate an estimate e that, while unrelated to the actual user estimate for this particular job, is expected to lead to the same general statistics of all the estimates taken together. To complete the model one just needs to note that Φ percent of the jobs are underestimated,[7] and for short jobs the estimates are too large by a factor of about 10 (accuracy of 10% or less). The final model is therefore

1. With probability of Φ return r (reconstructs the 100% peak).
2. Otherwise, create an estimate of r/u, where u is uniformly drawn from the range $(0, 1]$ (generates the uniform-like histogram).
3. If $r < 90$ seconds, multiply the estimate by 10 (recreates the hump near 0%).
4. If the estimate is "outrageous", truncate it to some upper bound.

Fig. 1.5 shows that, unfortunately, despite the added information, the Φ-model is also unsuccessful in capturing the "badness" of user estimates. In fact, one can always find a badness factor f that yields performance results that are closer to the real thing than when the Φ-model is employed.

Another model, similar to the Φ-model, was proposed by Cirne and Berman [20], which took the opposite direction in comparison to the previous model and chose to produce runtimes as multiples of estimates and accuracies, while generating direct models to the latter two. This decision was based on the argument that accuracies correlate with estimates less than they do with runtimes. In their model, accuracies were claimed to be well-modeled by a gamma distribution (a result of trying to model the uniform part of the histogram along with the hump at low accuracies, by using one function for both). Estimates were successfully modeled by a log-uniform distribution. This methodology suffers from the same problem as the previous model. In addition, it is not useful when attempting to add estimates to existing logs that lack them, or to workloads that are generated by other models which usually include runtimes and lack estimates [46, 30, 77, 99].

[7] Mu'alem and Feitelson did not use a parameter to denote the percent of killed jobs. Rather, they used a hard-coded value of 10%. In a later paper, Zhang et al. parameterized this value, called it Φ, and named the model accordingly: the "Φ-model".

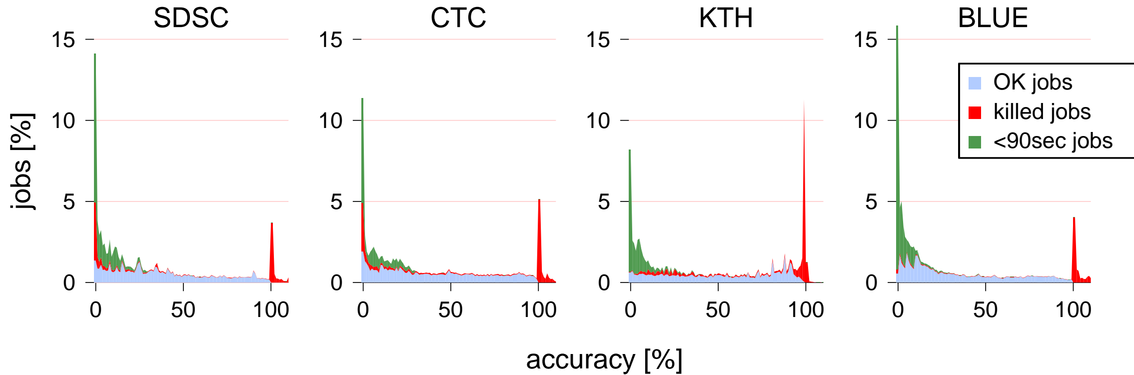


Figure 1.4: *The accuracy* $\left(= 100 \cdot \frac{runtime}{estimate}\right)$ *histograms are rather flat when ignoring jobs that reached their estimate and were killed by the system (100% peak) of failed on start up (0–15% hump).*

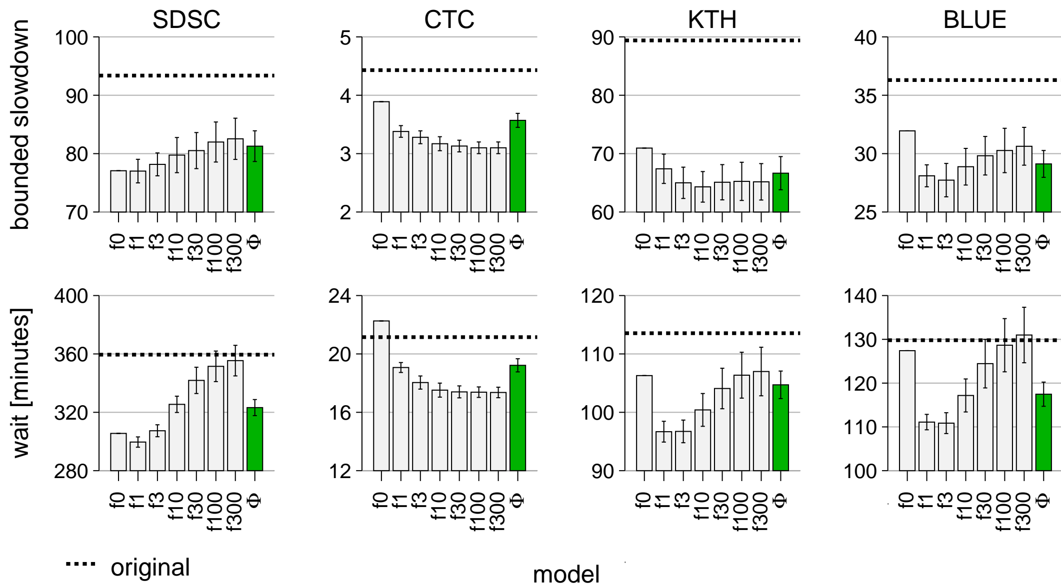


Figure 1.5: *Comparing the performance obtained when using real user estimates ("original") to those obtained when replacing them with artificial ones as generated by the f-model* ($f = 0, 1, 3, 10, 30, 100, 300$) *and the Φ-model.*

1.2.3 The Failure to Improve the Quality of Estimates for Backfilling

Prediction Algorithms There have been quite a few research efforts that attempted to present a higher quality alternative to user estimates. These have mainly focused on using historical data, as it is well known that users of parallel machines tend to repeatedly do the same work [48, 173] and therefore it is conceivable that historical data can be used to predict the future (Fig. 1.6). The common practice has been to partition past jobs into disjoint "similarity classes" or "categories" based on one or more of their attributes, including user and group ID, requested processors number, requested memory size, queue identifier, submit time, runtime estimate, executable name and arguments, and any other attribute which is known upon submission. When two historical jobs

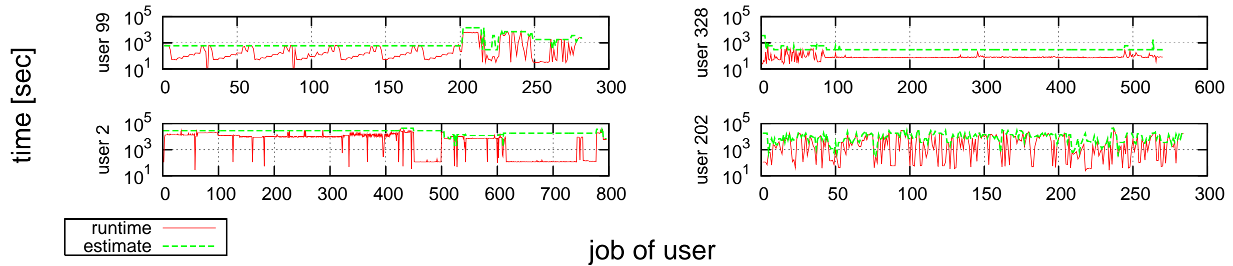


Figure 1.6: *Runtime and estimate of jobs by four arbitrary SDSC users show remarkable repetitiveness.*

agree on a predetermined subset of these attributes they are judged "similar".[8]

A chosen attribute subset, according to which jobs are partitioned into classes, is called a "template". When a new job arrives, the system checks whether its attributes, as listed in the chosen template, coincide with one of the similarity classes. If this is the case, the system generates a prediction based on the runtimes of the jobs that populate that class. The metrics used to generate the prediction (listed in order of increasing complexity) are: simply using the median or mean [136, 156], adding to the mean 1.5 standard deviations [108], using the top 95% confidence interval [62], using liner regression [135], using statistical models based on the (usually) log-uniform distribution of runtimes [31], and employing instance-based learning [83].

Prediction schemes may employ multiple templates simultaneously, in which case historical jobs may reside in more than one class. Consequently, new jobs may also match more than one class. When such a situation arises, the scheme must somehow determine which class to use for prediction. The canonical approach has been to use the one with the tightest statistical confidence [62, 135, 136, 86].

A fundamental problem underlying the approach described here is how to choose the templates according to which jobs are partitioned. The number of job attributes along with all their possible transformations is potentially very big. The number of possible templates is therefore exponentially bigger (recall that any subset of attributes and their derivatives can serve as a template), so it is obviously impractical to use all conceivable configurations. The simplest solution is to decide upon a static set of templates that seemed reasonable to the person that implemented the prediction algorithm [62, 108]. A more sophisticated approach is to do it dynamically at runtime, in an evolving manner. This is done based on the success that individual templates demonstrate in predicting future runtimes during execution, combined with some heuristic to periodically introduce new templates to replace those that are less successful, in the hope of converging to a template-collection that manages to produce robustly good results. This approach has been implemented using statistical methods [97], genetic algorithms [135, 136], and rough set theory [86].

The Failure of Prediction Algorithms Despite all the work devoted to prediction algorithms (denoted "predictors"), these have never found their way into production systems, and backfill schedulers in actual use still employ raw user estimates rather than history-based system-generated predictions. We identify three reasons underlying this failure:

[8]Agreement may be defined as simple equality, or if attribute values are equal only after undergoing some transformation, e.g. jobs sizes may agree if they fall in the same "range", submission times may agree if both are during daytime vs. nighttime, etc.

1. **Misperception of Estimates as Unimportant** As described above, many studies found that increased inaccuracy either doesn't degrade or even improves performance [146, 47, 169, 170, 34, 64]. This has led to the suggestion that estimates should be *doubled* [174, 108] or *randomized* [115], to make them even less accurate. These findings seemingly negate the motivation to incorporate mechanisms for better predictions, deeming user estimates as "unimportant".
2. **Complexity of Predictors** All previously suggested prediction techniques assumed that an important component in accurately predicting future runtimes is to identify the most similar jobs in the history, and base the predictions on them. This mandates logging and mining the history, often using very sophisticated algorithms. In addition, the metrics applied on the chosen historical jobs to produce the predictions have often been nontrivial by themselves. As described above, the end results has been relatively complex predictors employing various statistical methods [62, 31, 136, 97], genetic algorithms [136, 138], instance based learning [83], and rough set theory [86]. Further, they require a training period which can be significant. For example, Smith et al. [136, 138] used an entire trace to guide the selection of templates before evaluating their algorithm (on the very same trace, using the selected templates), deeming their algorithm as off-line and significantly limiting its practical value. Paradoxically, all this algorithmic and computational complexity is often much more complicated than the entire EASY scheduler, making existing predictors an unattractive alternative.
3. **Technical Barrier** While the above two difficulties are certainly contributing factors to the failure of predictors to find their way into production systems, they are not detrimental. The real problem is that it is simply impossible to naively replace estimates with system-generated predictions, because *estimates are part of the user contract*. While this contract clearly states that a job trying to exceed its estimate will be killed (con), it also guarantees that this job will be allowed to run until that time (pro). The difficulty arises because every reasonable prediction algorithm is bound to occasionally produce too-short predictions, leading to premature killing of jobs according to the backfilling rules, thereby violating the contract. Previous studies dealt with this difficulty using one of the following alternatives:

   - eliminating backfilling altogether, at the expense of fairness (using pure SJF) [62, 138]
   - employing speculative backfilling or test runs (assumes jobs are restartable) [115, 15, 90]
   - using preemption to e.g. suspend jobs exceeding their predictions and reinsert them to the wait queue (augments space slicing with time slicing) [62, 19, 139, 170]
   - considering only artificial estimates generated by the f- or Φ-models as multiples of actual runtimes (assumes underprediction never occurs) [174, 115, 15, 141, 142].

   None of these retain the appeal of plain EASY. Noting this problem, Mu'alem and Feitelson checked whether underprediction does in fact occur when using a conservative predictor (average of previous jobs with the same user, size, and executable, plus $1\frac{1}{2}$ times their standard deviation).[9] They found that ∽20% of the jobs suffered from underprediction and would have been killed prematurely by a backfill scheduler. They therefore concluded that [108]:

[9] In the terminology defined above, this is a template composed of the attributes $\{user, size, executable\}$.

THE UNFEASIBILITY CLAIM

> *"Given the large fraction of jobs that are underestimated, it seems that using system-generated predictions for backfilling is not a feasible approach".*

We will show that the above three difficulties can be either refuted, or dealt with in a simple and straightforward manner. Specifically, we will prove the unfeasibility-claim wrong.

The Inability of Users to Improve their Estimates An orthogonal effort to exploiting history has been recently conducted by Lee et al. [93], which explored whether users are able to improve their estimates if given enough incentive. The study involved users of the Blue Horizon supercomputer at the San Diego Supercomputer Center (denoted here as "BLUE"). In the hope to alleviate users' concerns that their jobs will be prematurely killed, users were given the opportunity to provide an additional runtime estimation value upon submission, such that their job would not be killed if this value is exceeded. Further, users were queried about their degree of certainty of the estimate they provide. Finally, to encourage users to do their best, a competition was declared where the most accurate user will win a prize. Results indicated that users rarely change their original estimate, are actually quite sure of themselves, and most probably would not be able to provide much better information.

1.2.4 The Problematic Nature of Raw Workload Data

The last issue we address in this dissertation is somewhat of a "meta issue". It is related to production workload logs and the manner in which these are used for research. For example, in this work we make heavy use of four such logs as the basis of most of our findings (SDSC, CTC, KTH, BLUE; to be introduced in the next chapter). However, the issue is much more general, and although its implications certainly have decisive consequences regarding the work presented here, it transcends the supercomputer domain and is actually applicable to all computer systems.

Importance of Representative Workloads It is well established that the performance of a computer system depends not only on its design and implementation, but also on the workload to which it is subjected [40]. Different workloads may lead to different absolute performance numbers, and in some cases to different relative ranking of systems or designs. Using representative workloads is therefore crucial in order to obtain reliable performance evaluation results.

The canonical way to obtain representative workloads is to use real workloads from production systems. One can record the workload on an existing system, and play back the recording to drive a simulation of a new system. If the existing system has a similar functionality to the new system being evaluated, one can assume that the same workload may apply. Likewise, if a new system design is shown to produce good results when applied to a wide range of such "recorded" workloads, one can claim the results are truly general. Indeed, this work, as well as numerous other papers, use this methodology and exploits the many workload logs that are freely available, for example, in the Parallel Workload Archive (from which the aforementioned four logs are taken [110]).

An alternative methodology is to use the recorded workloads as the basis for constructing a workload model (like in [77, 20, 99]), later to be used to generate input for a simulator of a new system. This has the benefit of allowing for more flexible usage, e.g. by modifying model parameters so as to adapt it to different system configurations.

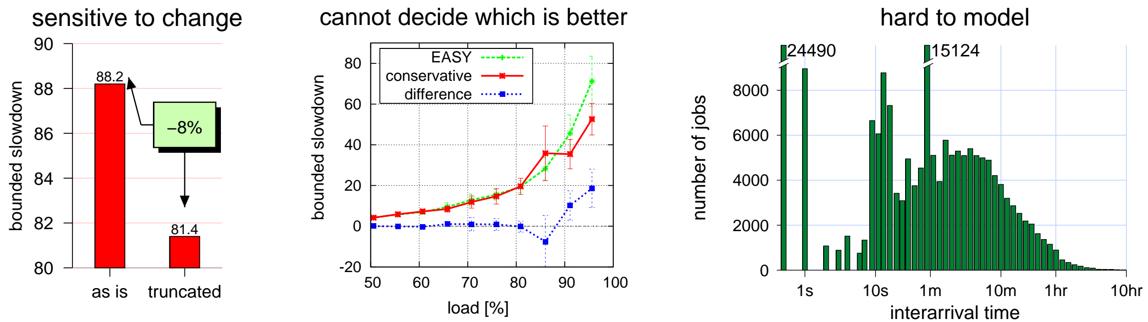


Figure 1.7: *Unwarranted implications of using raw data for performance evaluation and modeling.*

The Problem Using recorded workloads, however, has its problems. Consider modeling for example. This activity is typically done by collecting workload traces, and creating a statistical model based on fitting the distributions of workload attributes [89]. But such an approach is questionable if the data is not stationary. For example, Chiang et al. analyze six non-consecutive months of data from the NCSA Origin 2000 machine, and find that the workloads in different months may be quite different from each other [17]. It is also well known that workloads at different installations differ, and that workloads evolve with time as users learn to better use the system [70, 46].

In this context, we identify a different problem in basing an evaluation on production logs. Despite the overwhelmingly accepted view that real production workloads are representative and reliable, we claim that they may also contain anomalies that, while they do in fact occur, are actually non-representative *of the general case* and are therefore unreliable. The "general case" means the typical workload which is experienced by the relevant systems. We contend that one should exclude from the workload activities that are unique to a specific system when trying to evaluate the performance of systems in the general case. Our approach unfortunately contrasts common practices that view production logs as "the absolute truth", a situation we aspire to change.

Examples As a motivation for this topic, we briefly describe three examples demonstrating the types of problematic results we encountered while using raw data for performance evaluation and modeling. These are "real" examples in that we actually encountered them during the process of system evaluation. At the time, we were unaware these occur due to the aforementioned anomalies, and our perception was that we must learn how to live with such results, however undesirable.

The first example is depicted in the left of Fig. 1.7. It shows the average slowdown of *all* the simulated jobs under the EASY scheduler, when using two workloads. The first ("as is") is simply the SDSC raw log. The second ("truncated") is the very same log, but after we inject it with a negligible perturbation: we change the original runtime of a single specific job from 18 hours and 30 seconds, to exactly 18 hours. The 8% change in the *overall* average is overwhelming, as the perturbation is merely a 30 seconds change in the runtime of only one job *from tens of thousands of jobs that were submitted over a period of two years!* This instability a very disturbing and troublesome result, as it casts a serious shadow on the validity of what is maybe the most basic and standard performance evaluation methodology. Namely, if such a minor modification in the workload can trigger such a major change in the resulting average performance, who's to say that any relative ranking of systems and designs is not "bogus"? For example, if system A turns out to be better than system B only because some job terminated 30 seconds later, but the outcome would have otherwise been reversed, then this ranking is obviously completely meaningless. We will later show that this sensitivity is not representative of the general case.

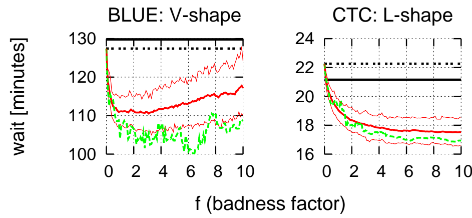


Figure 1.8: *Expressing performance in confidence intervals exposes clearer trends (compare with Fig. 1.3, page 10).*

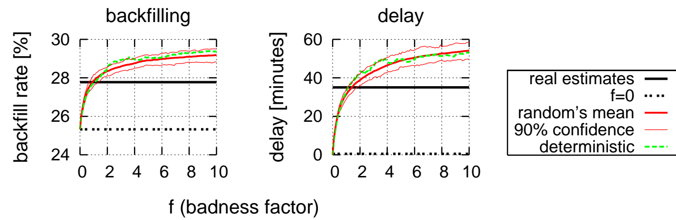


Figure 1.9: *Simulating CTC, the percent of backfilled jobs, as well as the average delay that non-backfilled jobs suffer due to inaccuracy, are monotonically increasing with f.*

Our second example is indeed along the lines of relative system ranking. Using the CTC workload, we compare the performance of two systems under various load conditions.[10] The evaluated systems are the EASY scheduler (a single reservation allocated to the first queued job) and the conservative scheduler (reservation allocated to all waiting jobs). Unfortunately, as shown in the middle of Fig. 1.7, the results are inconclusive: the answer to the question of which scheduler is preferable turns out to be dependent on the load. The bottom line is that the systems analyst is unable to make a clear recommendation, even though (we will later show that) one system is consistantly better than the other when stripping the workload form a non-representitive anomaly.

In addition to its impact on performance evaluation, using raw data has also negative implications on workload modeling. Demonstrating this, our third example deals with the distribution of interarrival times (elapsed time between submissions of consecutive jobs). The associated histogram of the LANL CM-5 log is shown in Fig. 1.7 (right). We find that the distribution is distinctly abrupt and modal, with several values that are extremely common (note the broken Y-axis). Importantly, this distribution cannot be fitted against (or modeled by) any standard distribution. We will show below that this is the result of aggregating the baseline workload with an anomaly.

1.3 Preview of Results

1.3.1 Solving the Mystery of Why Inaccuracy May Help

Sec. 1.2.1 introduced the inaccuracy mystery that, based on the popular f-model, prompted many researchers to claim that deliberately making estimates less accurate either improves performance, or has no real effect on it. These two observations were explained respectively by the contradictory "holes" argument (improvement is due to increased overestimation of long jobs that opens up larger holes for backfilling shorter jobs) and "balance" argument (no effect on performance because larger holes are cancelled out by backfill candidates appearing proportionally longer).

Why Performance Improves Indeed, we find that the average performance is extremely sensitive to minor changes in f and that the sample space is very noisy (Fig. 1.3). Thus, it is possible to conclude any of the two contradictory observations, if conducting only a small number of experiments in a non systematic manner. However, utilizing the random component of the f-model to perform repeated simulations and presenting the results in terms of mean and confidence[11] reveals a clear trend: the effect of increasing f is actually V or L-shaped, as is exemplified with BLUE and CTC in Fig. 1.8.

[10] The manner in which load is artificially varied is explained in the next chapter.

[11] Surprisingly, this was never done before.

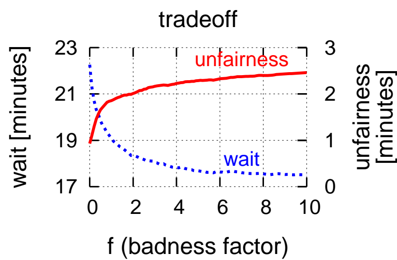


Figure 1.10: *Increasing f means trading off fairness for performance. (Results from simulating CTC.)*

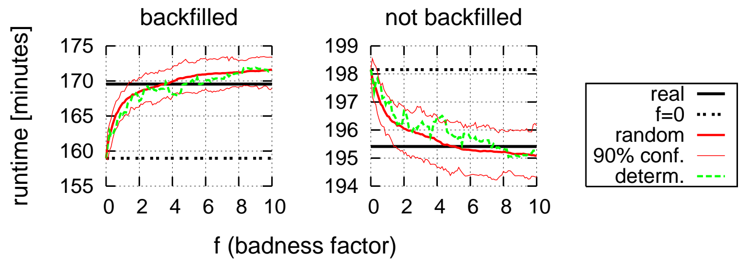


Figure 1.11: *The bigger f gets, the more the scheduler favors longer jobs for backfilling at the expense of shorter ones. (Results from simulating CTC.)*

We explain the descending part of the performance curves in this figure by reconciling the seemingly contradictory "balance" and "holes" arguments. In accordance with the "holes" argument, increased f does indeed mean more backfilling (Fig. 1.9, left). But the performance improvement is not just the result of this, as backfill candidates do in fact appear proportionally longer (in accordance to the "balance" argument). Rather, it is the result of what we call a "heel-and-toe" dynamic: a distinctive sequence of backfilling decisions that manages, step by step, to prevent the holes from closing up, leading to a preference for short jobs and the automatic production of an SJF-like schedule. On each step, the scheduler is "tricked" to believing the real earliest start time of the first queued job is further away in the future than it actually is. Fig. 1.9 (right) shows the average delay of jobs with reservations, beyond their hypothetical "correct" start times (had accurate estimates all of a sudden been made available to the scheduler, at the exact time instance when they became first in the wait queue and their shadow time was computed). The bottom line is therefore that multiplying estimates by a factor is actually *trading off fairness for performance*, as all the extra backfilling activity is at the expense of further delaying the jobs that have been waiting the most. This tradeoff if exemplified by Fig. 1.10, showing that the increasing "unfairness" of the schedule is a kind of mirror image to its improving performance.[12]

Why Performance Is Worsened We now go on to explaining the ascending part of the V-shaped performance curves (BLUE in Fig. 1.8). When f is very small, the holes in the schedule are relatively narrow, insuring only truly short jobs can enjoy them (explains the initial descending part of the curves). But as f becomes bigger, increasingly longer jobs fit the holes too, nudging the scheduler to "err" and favor longer jobs for backfilling at the expense of shorter ones. Fig. 1.11 demonstrates this pathology, showing that backfilled jobs become longer, while at the same time, non-backfilled jobs become shorter. The situation is worse for the random model, where long jobs can masquerade as short and vice versa. We analytically prove that the probability for this to occur is monotonically increasing as f goes to infinity. The absence of this random component from the deterministic model explains why it yields better performance than the random model (Fig. 1.8).

The remaining question is how come CTC's performance is L-shaped? Namely, how does it "manage to escape" the two aforementioned destructive processes? (Of longer jobs gradually fitting into the widening holes and of randomness.) Indeed, other logs are similar to BLUE and also correspond to V-shaped curves (not shown here). The solution to this mystery is related to load: the level and temporal structure of the activity exhibited in each log. It turns out that CTC is

[12]Unfairness is defined to be the average delay in the starting of jobs beyond what is "fair" (jobs that were started earlier than what is fair contribute zero to the average). The notion of "fairness" is accurately defined in Chapter 3.

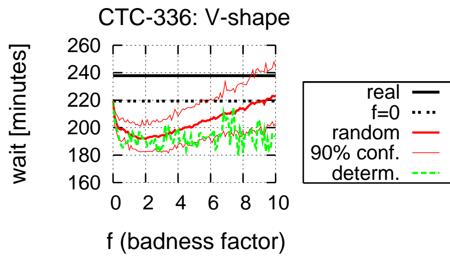


Figure 1.12: *Introducing burstiness to CTC by reducing the size of the simulated machine resulted in V-shaped performance curves, similarly to other logs (compare with Fig. 1.8).*

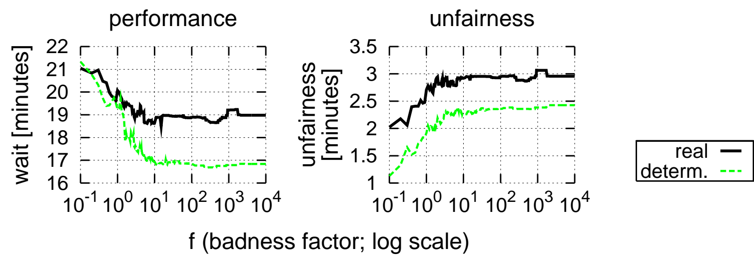


Figure 1.13: *The effect of multiplying real user estimates by a factor ("real") is qualitatively similar to using multiples of runtimes as estimates ("deterministic"), but the latter yields better results in terms of performance and fairness, meaning accurate estimates are better for backfilling. (Results from simulating CTC.)*

unique in that its load level is relatively stable, and in particular, unlike all the other logs, it is not bursty. Burstiness causes longer wait queues, a consequence of many jobs arriving at the same time. This intensifies both of the above destructive processes, as more backfill candidates with greater diversity allow the scheduler to make more "wrong" decisions. To demonstrate this, we simulated the CTC workload on a machine with less processors than the original system had: instead of 512 we used 336 (the size of the biggest job in CTC). This manipulation was verified to introduce burstiness, and indeed, the associated performance curves turned out to be V-shaped, similarly to all other logs (Fig. 1.12). Thus, performance curves are either V or L-shaped, depending on whether the workload is bursty or not, respectively.

Refuting the Myth The above observations that were obtained by using the f-model have seemingly only theoretical value, because we are multiplying actual runtimes, whereas this information is usually unavailable to the scheduler. However, there is one important practical implication. It turns out that all of our understandings regarding the effect of multiplying *perfect* estimates (runtimes), also apply when multiplying *real* estimates, as were given by users. Fig. 1.13 clearly shows this, as the trends exhibited by multiplying are qualitatively similar, regardless of whether the multiplied values are perfect or real (=flawed). The key is noting the quantitative difference. Apparently, in contrast to common belief, better accuracy does in fact improve performance in the sense that the more accurate the initial (to be multiplied) estimates are, the better the resulting performance becomes. As will be further discussed next, in no way does the act of multiplying emulate the inaccuracy exhibited by real users. Rather, it simply adds a certain "SJFness" to the schedule through heel and toe dynamics. Consequently, multiplying is actually not more than a (legitimate) scheduling policy that exercises the fairness/performance tradeoff. The bottom line is that system analysts should clearly distinguish between two types of inaccuracies:

1. **artificial** inaccuracy, which is generated by multiplying, trades of fairness for performance, and is a property of the scheduler, and
2. **real** inaccuracy, which is generated by and is a property of real users, and has the effect of worsening performance.

By no means is the first type adequate to serve as a model for the second. The problem is that, up till now, researchers confused between the two types of inaccuracies. This has led to the *false misconception* that "increased inaccuracy improves performance". The correct statement should be

that increased inaccuracy *worsen* performance, but that the scheduler can boost it up at the expense of fairness by multiplying the estimates with some factor.

This conclusion motives (1) developing an adequate model for real user estimates, and (2) improve the quality of estimates used by backfill schedulers. These issues will now be addressed in turn in the next two subsections, respectively.

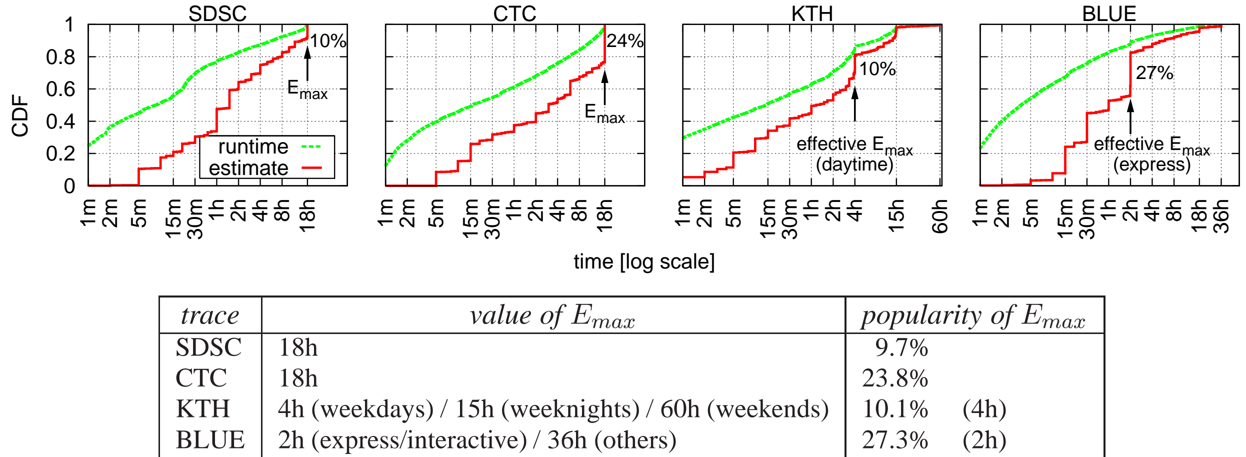
1.3.2 Modeling Estimates

Recall that Sec. 1.2.2 motivated the need for a realistic estimate model, and showed that the existing models (f and Φ) are inadequate. The bottom line was that using these models produces an unrealistic evaluation, whereby performance results turn out too good to be true (better than if real estimates were used). The previous Sec. 1.3.1 has shed some light on why this is the case (with respect to the f-model; in this section we will, among other things, do the same for the Φ-model). We now go on to focus on developing a better model.

Modality The fundamental and most important observation in achieving this goal is the following. Human users do *not* choose estimates that are uniformly distributed between the real runtime and its multiple with some constant factor, or anything similar. Rather, they use "round" estimates, like ten minutes, one hour, etc. In fact, we found about 90% of the jobs repeatedly use the same 20 "round" values. The result is a modal distribution, reflected in the staircase-like CDF curves shown in Fig. 1.14, in which each "mode" corresponds to a popular estimate. One particular popular value is E_{max}, the maximal estimate allowed. This value is a per-site administrative upper bound on estimates (and therefore on runtimes). The value of E_{max} is typically around 18h; in KTH and BLUE 4h and 2h serve as the "effective" E_{max} because most jobs were submitted during daytime or to the interactive/express queues, respectively. E_{max} is used by 10-27% of the jobs and is the most popular in three of the four logs (in SDSC it's ranked third). Its immense popularity can probably be attributed to (1) the fact underestimated jobs are killed by the system upon reaching their estimate, and (2) the inability of users to predict how long their jobs will run and their desire to "play it safe" and prevent their jobs from being prematurely killed.

Implications of Modality Regardless of why E_{max} is so popular, the implications are detrimental in terms of performance. This is true because any job with E_{max} as estimate will never be backfilled (as all the holes in the schedule are smaller than E_{max}, by definition). The more the jobs use E_{max}, the worse the performance gets. At the extreme, associating all jobs with E_{max} would mean backfilling activity (as depicted in Fig. 1.1) would completely stop and the schedule would largely revert to plain FCFS. Surprisingly, despite its decisive effect, the mere existence (and hence popularity) of E_{max} has been completely overlooked by past work, a fact that led to several mistakes.

One mistake is related to Fig. 1.15. Cirne and Berman hypothesized that the apparent connection between longer runtimes of jobs and improved accuracy is because the more a job progresses in its computation, the grater its chances become to reach successful completion [20]. Obviously, this hypothesis is false and unwarranted, because the existence of E_{max} guarantees long jobs to have high accuracy. For example, assuming E_{max} is 18h, if a job's runtime is 17h, then its estimate must be between 17h–18h (bigger than the runtime, smaller than E_{max}) and thus at least 94% accurate. In other words, long jobs are on the right of Fig. 1.15, where accuracy is high, while short jobs tend to be on the left, at lower accuracies.



trace	*value of E_{max}*	*popularity of E_{max}*
SDSC	18h	9.7%
CTC	18h	23.8%
KTH	4h (weekdays) / 15h (weeknights) / 60h (weekends)	10.1% (4h)
BLUE	2h (express/interactive) / 36h (others)	27.3% (2h)

Figure 1.14: *Top: cumulative distribution function (CDF) of runtimes and estimates. Unlike runtimes, estimates are modal. Runtime- are higher than estimate-curves, as runtimes are always shorter than estimates (underestimates jobs are killed). Bottom: per-site maximal allowed estimate and how many jobs use it.*

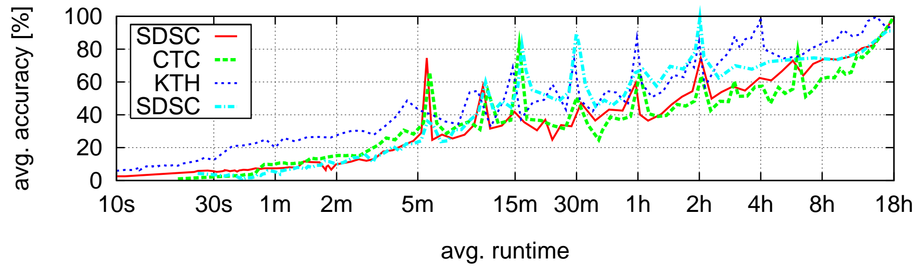


Figure 1.15: *Average accuracy* ($100 \times \frac{runtime}{estimate}$) *as a function of runtime. (The X-axis groups jobs to 100 equally sized bins according to their runtime.) Longer jobs enjoy higher accuracy.*

The same exact argument applies to another mistake, related to Φ-model (defined in Sec. 1.2.2 in an attempt to emulate the accuracy histograms shown in Fig. 1.4, page 13). Recall that the uniform part of the accuracy histograms was modeled by $e = r/u$, where u is uniformly distributed in $(0, 1]$, r is the runtime, and e is the resulting estimate. However, due to E_{max}, the distribution of jobs within the accuracy histogram is not at all uniform. Similarly to the previous mistake, here too (Fig. 1.4) long jobs must be on the right at higher accuracies, whereas only short jobs can reside on the left. Thus, E_{max}'s existence invalidates the rationale underlying the Φ-model.

Finally, we note that the harmful effect of modality is not just related to E_{max}. This is true because an estimate distribution that is dominated by only a few distinct modes (E_{max} and others) means less variance among waiting jobs, which means less flexibility for the scheduler to exploit existing holes (with various sizes) for backfilling.

Modeling Modality Our model therefore targets the modal nature of estimates. It relies on three input parameters: (1) the number of jobs composing the workload (namely, the number of estimates to produce), (2) E_{max}, and (3) the percent of jobs that use E_{max}. Based on these, instead of employing the common practice of artificially generating estimate values in isolation on a per-job basis, the model outputs a series of K modes given by $\{(t_i, p_i)\}_{i=1}^{K}$. Each pair (t_i, p_i) represents one mode, such that t_i is the estimate value (t stands for time), and p_i is the percentage of jobs that use t_i as their estimate (p for "popularity"). For example, CTC's series includes $(18h, 23.8\%)$,

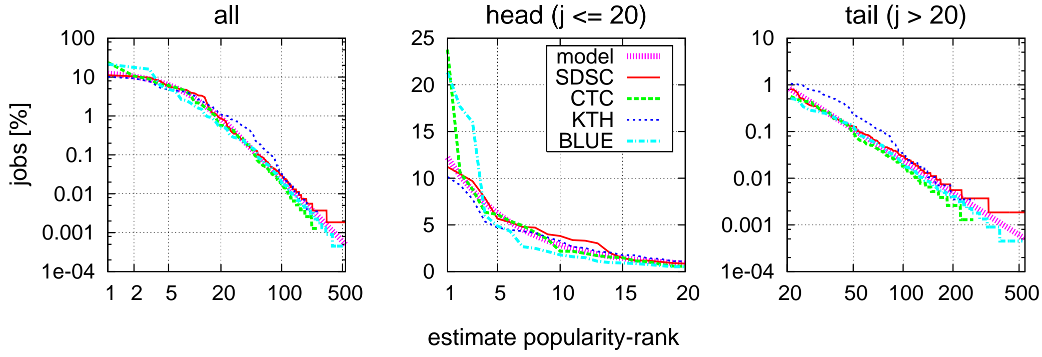


Figure 1.16: Given a head popularity rank $j \in [1..20]$, the associated percent of jobs is given by the $p_j = \alpha \cdot e^{-\beta \cdot j} + \gamma$ exponential function. Given a tail popularity rank $j \in [21..K]$, the associated percent of jobs is modeled by the $p_j = \omega \cdot j^{-\rho}$ power law. The middle figure has linear axes, while the other two are logarithmically scaled. The left figure concatenates the head and tail models.

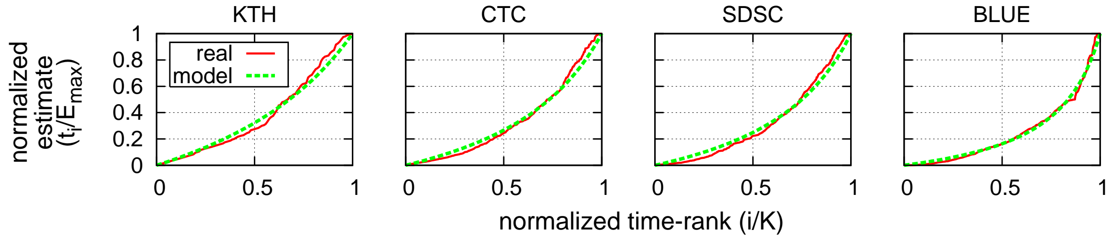


Figure 1.17: The smallest estimate is assigned a "time rank" of 1, the second most smallest has time rank of 2, etc. Normalizing a time rank means dividing it by K. Likewise, normalizing an estimate means dividing it by E_{max}. Given a normalized time rank $x_i = \frac{i}{K}$ ($i \in [1..K]$), the associated normalized estimate $y_i = \frac{t_i}{E_{max}}$ is modeled by the $y_i = \frac{(a-1) \cdot x_i}{a - x_i}$ fractional function.

because 23.8% of the jobs used 18 hours as their estimate. (After the series is constructed, our model offers a utility function to map the artificial estimates onto the jobs, such that each estimate is equal to or bigger than the associated runtime, as required by the backfilling rules.)

Our approach is to *separately* model the time values $\{t_i\}_{i=1}^{K}$ and the popularity values $\{p_j\}_{j=1}^{K}$, after which we define a mapping between times and popularities to create the pairs in the final mode series. But before doing this, we show that the modes naturally divide into two groups: the twenty most popular "head" estimate values (used by about 90% of the jobs throughout the entire trace), and the remaining "tail" estimates. We show that these two groups have distinctive characteristics. For example, Fig. 1.16 shows how the $\{p_j\}_{j=1}^{K}$ popularity series is modeled. The X-axis denotes the "popularity rank" j, where the most popular estimate has rank j=1, the second most popular has rank j=2, and so on. The Y-axis denotes the associated percent of jobs (=p_j). Indeed, "head" estimates are well modeled by an exponential function, whereas "tail" estimates obey a power law. In contrast, modeling the $\{t_i\}_{i=1}^{K}$ time series does not require the head/tail differentiation, and the entire series is successfully modeled by a fractional function (Fig. 1.17). Deciding which specific time values will serve as the twenty head estimates, however, requires a special effort.

Our conclusion from constructing the model is that users behave invariantly when it comes to estimating how long their jobs will run. Indeed, all modeled aspects of the estimates distribution are almost identical across all logs, allowing us to rely on only the three aforementioned input

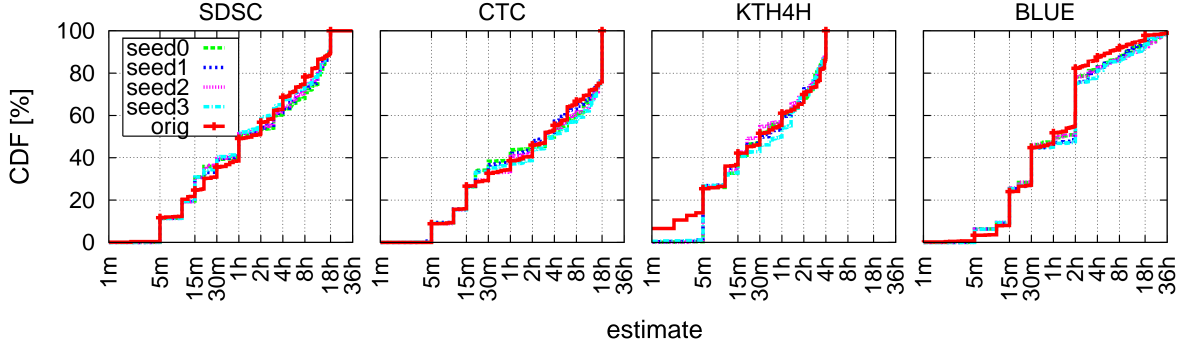


Figure 1.18: The original estimates distribution ("orig"; solid lines) is very similar to the modal output of our model, when used with four different seeds for the random number generator. (KTH4H contains only the "daytime" jobs.)

parameters, while still producing excellent results. Even though considerable variance does in fact exist, it is mostly encapsulated within the percentage of jobs that chose E_{max} as their estimate, which is indeed one of the three parameters. The remaining variance (if any) is attributed to another 1-2 very popular modes that sometimes exist, but are unique to individual logs. When provided this additional minimal information (optional parameters), our model's output is remarkably similar to that of the original (Fig. 1.18), but even with its vanilla setting results are satisfactory.

Finally, we show that when used in a simulation (by replacing real estimates with artificial ones), our model consistently yields performance results that are close to the original. The model is available for download at [155]. Unlike previously suggested models, it allows for realistic evaluation of the impact of increased inaccuracy on backfill algorithms, e.g. by systematically increasing the percent of jobs associated with E_{max}.

1.3.3 Incorporating System-Generated Predictions in Backfill Schedulers

Now that the source of the "badness" of user estimates is well understood, we go on to revisit the alternative: using history-based system-generated runtime predictions. In Sec. 1.2.3 we surveyed the considerable amount of work done on the subject and noted that, in spite of all this effort, predictors were never incorporated within production system. We identified three major difficulties underlying this failure, which we now revisit and resolve in turn, while outlining our contributions.

Addressing the Misperception of Estimates as Unimportant Recall that this difficulty emanates from many studies that found increased inaccuracy improves or doesn't effect performance [146, 47, 169, 170, 34, 64], yielding suggestions to make user estimates even less accurate by doubling [174, 108] or randomizing them [115]. This negated the motivation for improved predictors and implied accurate estimates are unimportant. We argue that this is false in three respects:

1. First, doubling (or multiplying) original user estimates indeed helps, but even more so if applied to perfect estimates (compare "realX2" to "perfectX2" in Fig. 1.2 and inspect the left of Fig. 1.13). We show that doubling of good system-generated predictions is similar: the more accurate the original predictions are, the more the doubling is effective.
2. Second, as mentioned above (Sec. 1.3.1), we show that the reason multiplying helps is due to a heel and toe dynamic, which allows shorter jobs to move forward within an FCFS setting

by implicitly approximating an SJF-like schedule. Recall that this is obtained by gradually pushing away the start time of the first queued job, effectively trading off FCFS-fairness for performance. One contribution of our work is showing that this tradeoff can be avoided to some extent by explicitly using a *shortest job backfilled first* (SJBF) backfilling order. By still preserving FCFS *reservation-order*, we maintain EASY's initial appeal and enjoy both worlds: a fair scheduler that nevertheless backfills effectively. We argue that in any case choosing SJBF is more sensible than "tricking" the scheduler to favoring shorter jobs by doubling estimates, randomizing them, or any other similar stunt.

3. The third fallacy in the "inaccuracy helps" claim is the underlying implied assumption that predictions are only important for performance. In fact, they are also important for various other functions. One example is advance reservations for grid allocation and co-allocation, shown to considerably benefit from better accuracy [83, 137, 96]. Another is scheduling moldable jobs (that may run on any number of nodes [31, 138, 22]). The scheduler's goal in this case is to minimize response time, considering whether waiting for more nodes to become available is preferable over running immediately. Thus, a reliable prediction of how long it will take for additional nodes to become available is crucial.

Addressing the Complexity of Predictors Three drawbacks were identified in previously suggested predictors (all algorithms suffer from at least one, and often all, of these drawbacks): suggested predictors are (1) based on identifying "similar" jobs in the history and therefore require significant memory resources and complex data structure to save the history of users, (2) they employ complicated prediction algorithms (to the point of being off-line), and therefore (3) pay the price in terms of excessive computational overhead spent on maintaining and mining the history [31, 62, 135, 136, 83, 86, 97]. In this context, our contribution is showing that a trivial predictor (free from all above drawbacks) can actually generate excellent results when used correctly, and explaining why this is the case.

The predictors we use are as simple as e.g. averaging the runtime of the two most recently submitted (and already terminated) jobs by the same user. Obviously, such an algorithm is very easy to implement and is almost overhead-free (it is simply a matter of saving two per-user numbers, updating them whenever a job by the user terminates, and averaging them out whenever a job by the user arrives). We show that when done correctly, this approach works very well. In fact, it turns out that for runtime predictions, *the recency of past jobs is actually more important than their similarity*. This is exemplified in Fig. 1.19, which is the summary of tens of thousands of simulations utilizing very many prediction algorithm variants. The algorithms differ in several respects, mostly related to the definition of the *history window*: which previous jobs to use to generate a prediction and how (exact details are provided in Chapter 4). One important parameter employed by all algorithms is the *window size*, which determines the number of past jobs the algorithm considers, such that the greater this number is, the more the predictor goes back in time to obtain this many jobs. The window size serves as the X-axis of Fig. 1.19. The Y-axis averages the performance degradation experienced by all the associated predictors, relative to some optimum (exact definition in Chapter 4). The conclusion is that the more history we exploit when making a prediction, the worse the resulting performance becomes. In fact, the performance degradation appears more or less linearly proportional to the size of the history window. This suggests that the considerable overheads of storing/mining the data for different classes of historical jobs [62, 136, 138, 83, 86, 96] are unwarranted, and that using only the 1-2 most recent jobs by the same

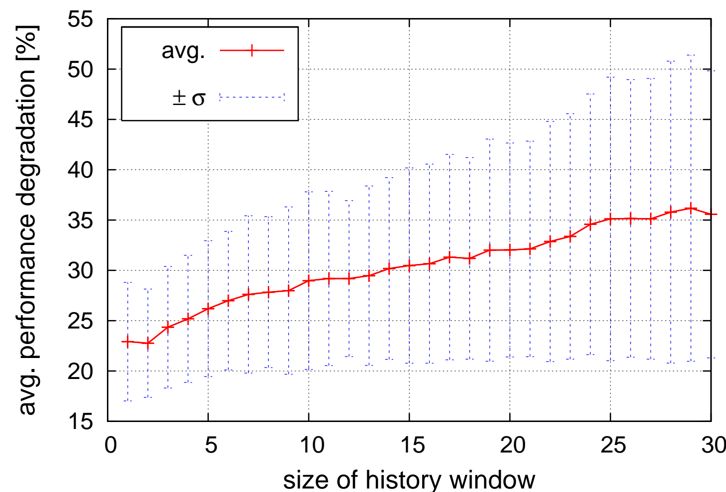


Figure 1.19: *Average performance degradation (± standard deviation) is more or less linearly proportional to the size of the window, indicating that using smaller windows with more recent information is preferable.*

user may be optimal. Arlitt et al. reached a similar conclusion in the context of the World Wide Web, contending that "only the topmost stack element is seeing significant reuse" when predicting the destination of a work-session based on the user's history stack [6].

Addressing the Technical Barrier Recall that what technically prevents systems from using system-generated predictions for backfilling is the issue of what to do when underprediction occurs. According to the backfilling rules, under-predicted jobs will be prematurely killed by the system, thus violating the contract with users that have requested to run longer than was predicted. Suggested solutions included simply ignoring the problem, using preemption, employing test runs, or eliminating the need for backfilling by using pure SJF [62, 174, 115, 19, 15, 90]. None of these retain EASY's initial appeal of simplicity and fairness. Mu'alem and Feitelson checked the extent of the underprediction phenomenon, showed it to be significant (20% of the jobs), and concluded that "it seems using system-generated predictions for backfilling is not a feasible approach" [108] (this statement was termed the "unfeasibility claim").

The initial part of our solution to this problem is noticing that estimates have a dual role: (1) to serve as runtime approximations, and (2) to serve as kill-times. We argue these should be separated. As it is legitimate to kill a job once its user estimate is reached, but not any sooner, the main function of estimates is in fact to serve as kill-times. At the same time, there is nothing to stop us from basing all the other scheduling considerations on the best available information.[13]

Our first step in utilizing predictions for backfilling is therefore doing just that, namely, basing everything, but kill times, on predictions instead of estimates.[14] This means the reservation time is computed based on predictions of running jobs. Likewise, waiting jobs that serve as backfill candidates are judged suitable for backfilling only if their prediction indicates they will terminate before the reservation time. This optimization has the positive effect of dramatically improving the

[13]Note the terminology: "estimate" refers to the runtime approximation provided by the user upon job submittal, whereas "prediction" refers to the value that the system eventually uses. A prediction can be set to be the estimate, but it can also be automatically generated by a predictor, as described here. However, "prediction" is an overloaded term: when "estimates" and "predictions" are contrasted, the latter actually means "system-generated predictions".

[14]Recall our predictor generates a job's prediction e.g. as the average runtime of the last two jobs by the same user.

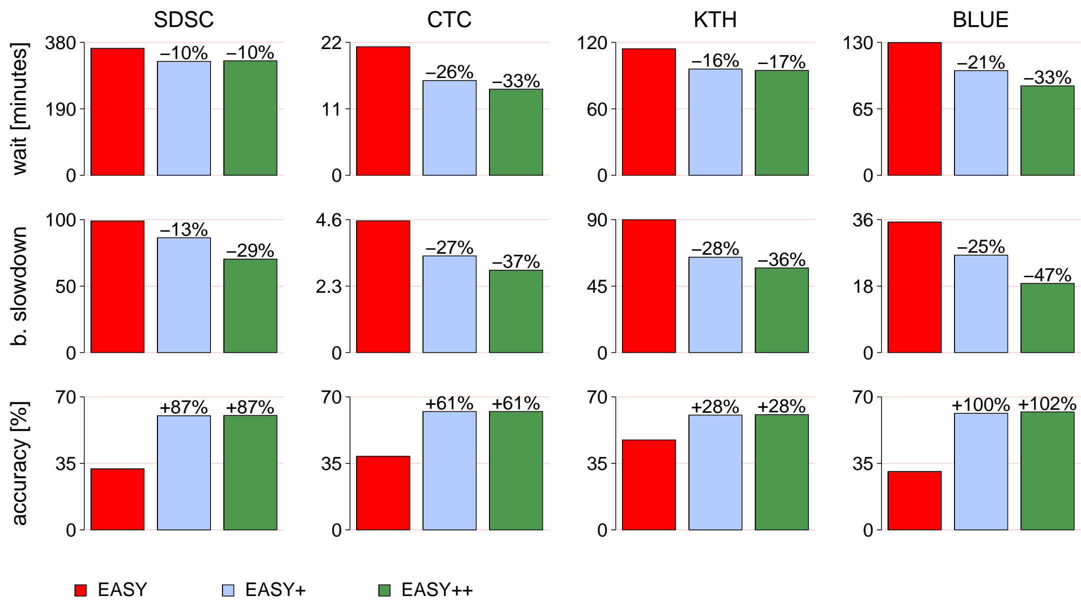


Figure 1.20: *Comparing average performance and accuracy of the EASY, EASY^{+}, and EASY^{++} schedulers. The numbers at the top of the bars show the associated percentage change relative to EASY: negative values indicate an improvement of the performance metrics (wait/slowdown); positive values indicate an improvement of accuracy.*

average accuracy. Regrettably, in spite of this improvement, performance is worsened by up to an order of magnitude! A thorough analysis revealed that this is due to "expired" predictions, outlived by their jobs. Such stale information leads the scheduler to erroneously believe that processors of currently running under-predicted jobs should be available at the present time. The result is an unrealistically too close shadow time that opens up a very small hole in the schedule for backfilling. At the extreme, the reservation is made for the present time effectively stopping all backfilling activity (as shown in Fig. 1.1) and degrading the scheduling towards plain FCFS.

One way to tackle this problem is to try to minimize it by producing more "conservative" (bigger) predictions. But as mentioned above, this alternative was shown to be unsuccessful (20% of the jobs prematurely killed [108]). Further, just "minimizing" the problem is simply not enough, as the contract with users should *never* be violated. We therefore propose a simpler alternative that finally manages to provide a solution.

The basic idea is to avoid the issue altogether, by refraining from placing the burden of handling underprediction on the *predictor*. Instead, we modify the *scheduler* to dynamically increase expired predictions proven too short. The underlying rationale for this is the following. If a job's prediction indicated it would run for ten minutes, this time has already passed, but the job is still alive, why not do the sensible thing and accept the fact it would run longer? Thus, when underprediction is detected, we acknowledge the fact the user was smarter than us and set the new prediction to be the original estimate as was given by the user. Once the prediction is updated, this effects reservations for queued jobs and re-enables backfilling.

Fig. 1.20 shows some of the results of our approach. EASY^{+} replaces user estimates with predictions, while employing prediction correction. The typical improvement over vanilla EASY is around 25%. EASY^{++} adds SJBF to EASY^{+} and demonstrates an additional improvement.

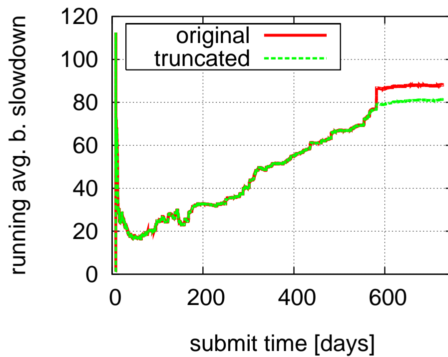


Figure 1.21: *Comparing the running average slowdown reveals the difference in the performance occurs in the 581st day.*

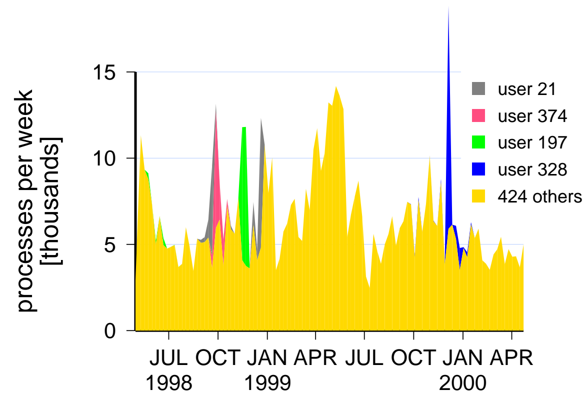


Figure 1.22: *In the 581st day, user 328 generated a huge flurry of more the 10,000 submitted processes, aggregated to 32-sized jobs that react en masse to change.*

This can up to double the performance relative to the baseline (BLUE/slowdown). Finally, under both EASY^{+} and EASY^{++} the average accuracy stabilizes at a bit more than 60% (which is again double the baseline in the case of BLUE).

1.3.4 Workload Flurries and Sanitization

Recall that Sec. 1.2.4 introduced three problematic examples that arose during the process of workload modeling and system performance analysis. Specifically, these involved the sensitivity of overall performance to negligible change, inconclusive system ranking, and the failure to fit a workload attribute using a standard distribution (Fig. 1.7, page 17). As will be described next, we discovered that all of these were the result a previously unknown phenomenon we call "workload flurries" (Chapter 6 discusses various types of anomalies): rare surges of exceptionally large repetitive activity, generated by a single user, that dominate the system for a limited period of time.

Impact of Flurries Focusing on the first example (minus 30 seconds in the runtime of one job leads to 8% change in overall performance), we define the *running average slowdown* at time T to be the average slowdown of all those jobs submitted prior to T. When plotted as a function of submission time, this metric exposes how the average performance evolves. Examining the associated graph (Fig. 1.21), we see that most of the 8% difference opens up at the 581st day. Further examination reveals that the cause of this difference is a group of a few hundred big jobs with identical attributes that were submitted by a single user. Fig. 1.22 pinpoints this group (rightmost peak); as mentioned earlier, this type of an activity is called a workload flurry. In a nutshell, the fact all the jobs within the flurry have the same characteristics makes them tend to be similarly affected by change. Such a tendency has a decisive effect on average performance, because the manner in which the first job reacts to the initial change is tremendously amplified by all the succeeding jobs in the flurry that react similarly, collectively "pulling" the entire average in the same direction.

The graph shown in Fig. 1.22 is typical to relatively long logs, as most that are longer than a year have at least one flurry in them. Once we discovered a flurry was to blame regarding the sensitivity issue, it became clear that this is not an isolated incident. Fig. 1.23 reevaluates our three examples using "cleaned" workloads from which flurries were excluded. As we can see, the

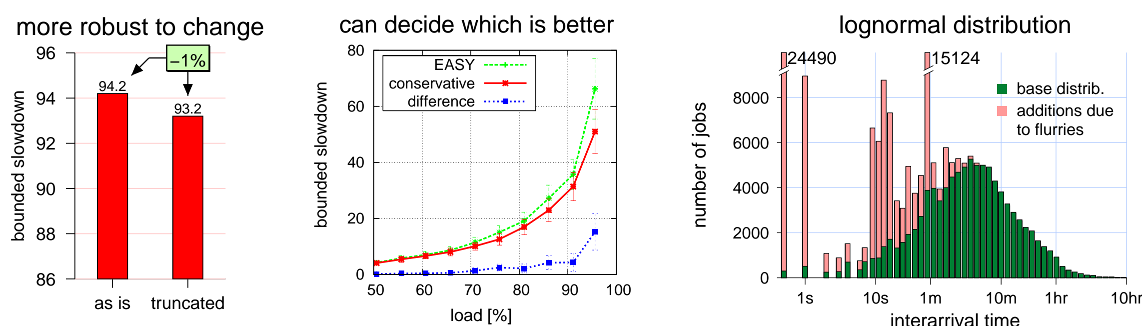


Figure 1.23: *Removing flurries from the workloads and repeating the experiments using the cleaned versions makes our troubles go away (compare with Fig. 1.7, page 17).*

results are more robust to change (left), a consistent trend emerges when comparing the schedulers (middle), and the examined distribution turns out to simply be lognormal (right).

We note that the flurries phenomenon is not unique to the supercomputing domain and was also identified in other types of systems for which we had long logs available (various departmental servers including file servers, authentication servers, and CPU servers).

Data Sanitization for Reliable Modeling We argue that modeling activity conducted naively on the basis of raw data which contains flurries is methodologically erroneous. It is in fact a lose-lose situation, whereby both "normal" activity and the flurries themselves are modeled incorrectly. Consider the interarrival distribution example in Fig. 1.23 (right). The abrupt flurry modes associated with shorter values actually occur within a very limited period of time. A general fit which is oblivious to this fact will evenly disperse these values through the entire workload, thus (1) failing to recreate the intense temporal flurry activity, and (2) ruining the otherwise lognormal structure of the background normal conditions with the irregular monolithic modes. A better methodology is to first divide the workload into "normal" conditions and "flurries", and then to separately apply current methodologies on the different parts. Modeling flurries, however, in the current status of things, is not a generalized approach, as no known flurry is representative of another.

Data Sanitization for Reliable Performance Evaluation Flurries are non-representative of the typical parallel workload in that they are unique, rare, and temporally confined events, which do not occur in the normal mode of work. In addition, flurries are also non representative of each other, as each flurry has its own distinctive characteristics. Since most of the time the workload is flurry-free, the results of an evaluation that deletes flurries from the raw workload will likewise be applicable most of time. This approach is aligned with the standard methodology employed in computer architecture research to use short "stitched" versions of a standard benchmark applications instead of the actual benchmarks, in the interest of reducing simulation time (a stitched version is composed of several "representative" fragments of the application's instruction stream that are concatenated together) [114, 117].

A stronger justification for the removal of flurries is that they actually have *no* affect on the performance metrics associated with the baseline activity. Rather, the simulation instability reported earlier is merely the artifact of aggregating the metric values of flurry and non-flurry jobs within the same average (slowdown in our case). Separating this monolitic average into two values — one that averages only flurry jobs and another that averages all the rest — reveals that the latter

is actually stable, well-behaved, and insensitive to minor changes; the former, on the other hand, is exactly the opposite and is responsible for the observed instability, being extremely sensitive to minor perturbations. Note that presenting a perfromance metric as a set of several averages, such that each is associated with different group of jobs, is also a standard methodology (narrow jobs vs. wide, short jobs vs. long, etc.) [125, 115, 141].

The non-flurry average has *significant* value: it embodies the system performance *as experienced by the vast majority*. Factoring in the flurry average within the same number merely introduces unwarranted noise that distorts the underlying result. In contrast, the flurry average has no meaning other than examplifying the sensitivity of the associated jobs. (One certainly cannot use it as a representative quantity, as it can change by orders of magnitude when negligible perturbations are introduced.) Importantly, we find that the effect of separating the average performance into two numbers (and considering only one), or deleting the flurry from the raw workload altogether, is qualitatively similar. We therefore recommend the latter alternative since it is clearly the simpler one. Indeed, with a sanitized version of the log, any analyst can simply and immediately use the log, whereas with the raw version, the analyst (1) must be aware of all the details and (2) must make a repeated effort to separate the population to flurry vs. non-flurry jobs.

Data sanitization conflicts with certain common views, which consider the original workload as almost "sacred" (a popular view in the computer systems community is that if one "tampers" with the data, one might as well arbitrarily decide upon the results in any way one chooses). But this is in disagreement with what is routinely done in every sound statistical analysis, where data is throughly validated and, if necessary, cleaned (removal of outliers). So sanitization is certainly not a far fetched idea. Moreover, to argue for an evaluation based on workloads with flurries, one must argue that the activity of a specific user during a relatively very short time should indeed dominate the evaluation results. Also, one must be satisfied with results that change considerably and even swing the other way if conditions are slightly changed or the span of the examined time covered by the evaluation is shifted such that the flurry is excluded. If flurries are removed, one at least may argue that the evaluation result are correct, say, 95% of the time, and in any case to the vast majority of the jobs; no such claim can be made if they are left in.

Once again, as noted with respect to the modeling process, we advocate a two-phase evaluation: with flurries first excluded and later included. However, here too the fact no flurry is representative of another limits the generality of the results of the latter. A real generalization (if possible) is left for future work, to be conducted when more date and knowledge related to flurries are collected.

Dissemination of Data Finally, finding flurries is a nontrivial task. We will later show there are other types of non-representative data, and these are also hard to find. Indeed, numerous papers have used the aforementioned logs (and others) in their raw form for performance evaluation and modeling, oblivious to the fact the various anomalies that the logs contain compromise the results. We therefore contend that data should be shared along with all the accumulated related information. As a first step, we have updated the Parallel Workload Archive, which is the source of the above logs, to include a cleaned version of the logs, in addition to the raw version [110].

Chapter 2

Methodology

This chapter describes our methodology, as used throughout the rest of this dissertation.

2.1 The Trace Files

All of our results are based on simulating and modeling workload logs from real production system. These are available through the PWA (Parallel Workload Archive) [110] in a standardized format called SWF (Standard Workload Format) [147]. Briefly, logs are given in plain ASCII text. The top of each such log file contains "header comments" that describe (through standardized fields) the general aspects of the respective workload: which machine generated it, how many processors it has, what are its queues and partitions, etc. The body of the log is a sequence of lines, such that each line represents a job. A line is composed of eighteen fields (as defined by the SWF) separated by whitespace. Each field specifies a job attribute: the job's arrival time, runtime, estimate, processors number (size), user, group, memory size, completion status, executable, queue, partition and more. The SWF dictates that all valid attribute values must undergo a transformation to be expressed as nonnegative decimal numbers. However, if an original value is missing or corrupted, the standard states it should be set to -1.

The logs we have used in this dissertation are listed in Tab. 2.1 (these are most of the PWA's logs; the missing ones were added too recently to be usable in this context). The four top logs are the ones containing data about real estimates. Only these four are used in Chapters 3-5; the remainder are used solely in Chapter 6. Note that the specified data relates to the original trace files ("raw"), their "cleaned" version (which the PWA recommends to use), and a "sane" version. The first two can be freely downloaded from the PWA. The sane version applies a filter on cleaned logs, removing all jobs that cannot be used in simulations due to missing size, runtime, or submission-time information. Chapter 6 actually constitutes the basis for the PWA's recommendation to favor cleaned logs over their raw form. Indeed, Chapters 3-5 use the sane versions only, whereas Chapter 6 uses raw logs solely for the purpose of establishing the case against them.

Finally, note that each log file is associated with a version number of the raw log, and possibly of a cleaned log ("cln"). If the latter is missing, this means no anomalies or non-representative activities were found within the original log. Therefore, the raw log version is also considered to be the clean version. The version number itself reflects a specific instance of the log.[1] This is

[1] A new version for a raw log is created only when identifying a problem in the conversion process (from the

abbreviation	*site*	*machine*	*version*		*cpus*	*duration*			*load*	*job number*			*avg run time [min]*
			raw	*cln*		*start*	*end*	*mon -ths*		*raw*	*clean*	*sane*	
CTC	Cornell Theory Ctr	SP2	1	1.1	512	Jun 96	May 97	11	56%	79,302	77,222	77,222	188
KTH	Swedish Royal Instit. of Tech.	SP2	1		100	Sep 96	Aug 97	11	69%	28,490	28,490	28,490	148
SDSC	San-Diego Supercomput. Ctr	SP2	2	2.1	128	Apr 98	Apr 00	24	84%	73,496	59,725	54,053	123
BLUE	San-Diego Supercomput. Ctr	Blue Horizon	2	2.1	1,152	Apr 00	Jan 03	32	76%	250,440	243,314	223,407	73
NASA	NASA Ames Research Ctr	iPSC/860	1	1.1	128	Oct 93	Dec 93	3	47%	42,264	18,239	18,239	13
LANL-CM5	Los Alamos National Lab	Conn. Machine	2	2.2	1,024	Oct 94	Sep 96	24	74%	201,387	122,055	122,052	43
SDSC-Par95	San-Diego Supercomput. Ctr	Paragon	1	1.1	416	Dec 94	Dec 95	12	68%	76,872	53,947	53,133	68
SDSC-Par96	San-Diego Supercomput. Ctr	Paragon	1	1.1	416	Dec 95	Dec 96	12	72%	38,719	32,136	31,334	138
LLNL-T3D	Lawrence Livermore Nat. Lab	Cray T3D	1		256	Jun 96	Sep 96	4	62%	21,323	21,323	21,323	23
SDSC-SP2	San-Diego Supercomput. Ctr	SP2	3	3.1	128	Apr 98	Apr 00	24	84%	73,496	59,725	54,051	123
LANL-O2K	Los Alamos National Lab	Origin 2000	1		2,048	Nov 99	Apr 00	5	64%	121,989	121,989	116,996	86
OSC	Ohio Supercomput. Ctr	Linux Cluster	1		178	Jan 00	Nov 01	22	51%	80,713	80,713	80,713	220
DAS-Amst	DAS2 Grid Amsterdam U.	Linux Cluster	1		64	Jan 03	Dec 03	12	20%	66,429	66,429	65,381	20
DAS-Leiden	DAS2 Grid Leiden U.	Linux Cluster	1		64	Jan 03	Dec 03	12	12%	40,315	40,315	39,356	18
DAS-Vrije	DAS2 Grid Vrije U.	Linux Cluster	1		144	Jan 03	Jan 04	12	15%	225,711	225,711	219,618	9
DAS-Delft	DAS2 Grid Delft U.	Linux Cluster	1		64	Jan 03	Dec 03	12	11%	66,737	66,737	66,112	12
DAS-Utrecht	DAS2 Grid Utrecht U.	Linux Cluster	1		64	Feb 03	Dec 03	11	14%	33,795	33,795	32,953	47

Table 2.1: *Real production logs used as the basis of this study. See the PWA for more details [110].*

specified to promote the reproducibility of results, as all instances are available through the PWA.[2] Specifically, note that the abbreviations "SDSC" and "SDSC-SP2" refer to different versions of the same log. Chapters 3-5 use the former, while Chapter 6 uses the latter (the difference, however, is negligible). This is the only occasion in which a log appears twice in Tab. 2.1, meaning it presents 16 different logs (rather than 17).

2.2 The Simulator

The performance evaluation done in this work is based on an event-based simulation of the respective system. This is basically EASY scheduling, with possible modifications as noted in the context in which the evaluation is conducted (e.g. changing the scanning order of jobs for backfilling). Events are job arrivals and terminations (Chapter 4 adds an additional event, to be described in the relevant context). Upon arrival, the scheduler is informed of the number of processors the job needs and its estimated runtime. It can then start the job's simulated execution or place it in a

original format of the log to SWF). A new version for a cleaned log is created when an additional non-representative anomaly is discovered. The major version number of the cleaned version associates it with a raw log, such that some filter was applied to the latter in order to form the former.

[2]Naturally, one cannot expect from researchers to redo all their work whenever a problem with one of the logs is encountered.

queue. Upon a job termination, the scheduler is notified and can schedule other queued jobs on the free processors. Job runtimes are part of the simulation input, but are not given to the scheduler.

The input used to drive simulations are the topmost four SWF files (those with user estimates data), as listed in the Tab. 2.1. Since these traces span the past decade, were generated at different sites, on machines with different sizes, and reflect different load conditions, we believe consistent results obtained in this work are truly representative. The logs are simulated using the exact data provided, with possible modifications as noted (e.g. to check the impact of replacing user estimates with system generated predictions).

2.3 Simulating EASY Backfilling

The EASY backfilling algorithm was briefly described in the previous chapter. In this section we provide a more detailed description. The scheduler responds to two types of events: job arrival (also denoted job submission) and job termination. The handling of a job's arrival is

1. insert the job into the FCFS waiting queue, and
2. invoke the schedule function (that starts as many waiting jobs as possible while obeying the backfilling rules),

The handling of a job's termination is

1. increase the *capacity* of the machine (the number of currently availablty free processors) by the size of the terminated job, and
2. invoke the schedule function.

The implementation of the schedule function is as follows:

1. Let the FCFS waiting queue be denoted as Q. If Q is empty, return.
2. Let the job at the head of Q be denoted J — this is the longest waiting job. Let $J.size$ be the number of processors required by J. Let C denote the current capacity of the machine. If $J.size \leq C$ then
   (a) update $C = C - J.size$,
   (b) remove J from Q,
   (c) start executing J, and
   (d) go to step 1.
3. Now that we are sure that Q is not empty and that J is too big to be started ($J.size > C$), start the backfilling process:
   (a) Find the "shadow time" (also denoted "reservation time") and the "extra nodes":
      i. Sort the list of running jobs according to their estimates termination time (this is the start-time plus the user runtime estimate).
      ii. Iterate through the list of running jobs and accumulate nodes until their number is equal to or bigger than $J.size$.
      iii. The (estimated) time at which this happens is the *shadow time*.

   iv. If, at this time, more nodes are available than needed by J (the first queued job), the ones left over are the *extra nodes*.

(b) Find as many jobs to backfill as possible by traversing Q (in FCFS order) and checking for each job whether one of the following two conditions hold:

   i. It requires no more than C nodes and will terminate by the shadow time, or

   ii. It requires no more than the minimum of C and the extra nodes.

(c) If either of the two conditions are met, remove the job from Q, start it, and update C accordingly. Additionally, if the job was backfilled at the expense of the extra nodes, reduce the number of the extra nodes by the size of the backfilled job.

Backfilling against the shadow time is illustrated in Fig. 1.1 (page 5). Backfilling at the expense of the extra nodes is illustrated in Fig. 3.14 (page 49).

2.4 Performance Metrics

Like most related studies [51], we measure the performance of systems using two metrics: average wait-time (A_{wait}) and bounded slowdown (A_{bsld}), where the average is taken over the jobs that participate in the simulation.

A job's *wait time* is defined to be the duration of the period between its submittal and starting time (in this work we usually use minutes to express this metric). Related studies sometimes prefer to use *response time* (instead of wait time), defined to be $w+r$, where w and r are the job's wait and running time, respectively. However, A_{wait} is preferable over average response time ($A_{response}$), because for batch systems the difference between the two is a constant, regardless of which batch scheduler is being used. The constant that forms the difference is actually the average runtime ($A_{runtime}$), because

$$A_{response} \equiv \frac{1}{n}\sum_{j\in J}(w_j + r_j) = \frac{1}{n}\sum_{j\in J} w_j + \frac{1}{n}\sum_{j\in J} r_j = A_{wait} + A_{runtime}$$

where J is a set containing all participating jobs, n is its size, and w_j and r_j are the wait-time and runtime of job j, respectively. Since $A_{runtime}$ is a given that is unaffected by the scheduler, preferring wait-time implies focusing only on the scheduling activity and neutralizing the highly variable average runtime (rightmost column in Tab. 2.1).

The *slowdown* metric is response time normalized by running time: $\frac{w+r}{r}$, reflecting the relative delay factor of the job (e.g. if a job's runtime is one hour and it had waited for two hours before being scheduled, then it suffered from a slowdown of 3; the optimum is of course 1). The *bounded-slowdown* metric eliminates the emphasis on very short jobs (e.g. with zero runtime) due to having the runtime in the denominator; a commonly used threshold of 10 seconds was set yielding the formula:

$$\textit{bounded slowdown} = \max\left(1\ ,\ \frac{w+r}{\max(10, r)}\right).$$

Finally, to reduce warmup and cooldown effects of the simulation, the first 1% of terminated jobs and those terminating after the last arrival were not included in the metrics averages [76]. In the above definition of response time, for example, this means that these jobs were excluded from

the J set, despite the fact they actually participated in the simulation. This is true not just for the two performance metrics defined above, but also to the very many other metrics we introduce in their respective context (e.g. the backfilling rate of a schedule).

2.5 Artificially Varying the Load

In this dissertation, the term "load" is a synonym to what is often referred to as "offered load" or "utilization". Under the constraint of avoiding the warmup/cooldown effects, load is defined to be

$$load = \frac{\sum_{j \in J} size_j \times runtime_j}{P \times (T_{end} - T_{start})}$$

where P is the number of processors composing the parallel machine, T_{start} is the last termination within the 1% aforementioned jobs, and T_{end} is the last arrival. Thus, load is a fraction in [0,1], or in [0,100] if expressed in percents. Note that the above formula is a simplification: every job that ran within the $[T_{start}, T_{end}]$ time period contributes to the numerator, such that the runtime is replaced with a value which is truncated according to the upper and lower bounds of this period. In other words, the term $runtime_j$ is replaced with

$$\min(T_{end}, termination_j) - \max(T_{start}, start_j).$$

When systems are underloaded, their performance is typically very similar (e.g. there can be very little difference between FCFS and EASY). Higher load conditions expose the real differences in how systems perform. In this work we therefore often manipulate the log files to reflect different load conditions. The standard way to do is to multiply the job interarrival times by a constant. For example, if the original offered load is ρ, multiplying all interarrival times by a factor of $\rho/0.8$ will change the offered load to 0.8.

Chapter 3

Backfilling Dynamics: Solving the Mystery of Why Increased Inaccuracy May Help

3.1 Introduction

Context This chapter was fully introduced in Sec. 1.2.1 (page 9) that also conducted a detailed survey of related work. Briefly, this chapter builds on the many studies that researched the impact of inaccurate estimates on the performance of backfill schedulers. The de-facto standard for doing this was modeling estimates by the "f-model" [146, 47, 174, 169, 108, 15, 142, 170, 122, 34, 64], where $f \geq 0$ is a "badness factor", such that given a runtime r, the associated estimate is uniformly distributed in $[r, (f+1) \cdot r]$ (the "random model"), or is simply set to be $r \cdot (f+1)$ (the "deterministic mode"). With this, increasing f translates to increased inaccuracy.

Surprisingly, inaccurate estimates ($f > 0$) yielded better performance than accurate ones ($f = 0$). Additionally, some researchers observed that this improvement is largely insensitive to the exact value of f, while others suggested bigger fs imply a bigger improvement. Indeed, due to the noisy nature of results, both observations are possible if using only a few experiments in a non-systematic manner (Fig. 1.3, page 10). The fact $f > 0$ improves performance was unanimously explained by the *holes argument*, claiming bigger fs imply wider holes in the schedule that allow for more effective backfilling [47, 174, 108, 15, 142]. In contrast, the observed "insensitivity" of performance to the exact $f > 0$ value was explained by the *balance argument*, claiming the effect of bigger holes is cancelled out by backfill candidates appearing proportionally longer (as their runtime is multiplied by f too) [174, 169, 170, 64]. We noted that while both arguments make sense, they are contradictory, and in any case fail to explain the trends observed in Fig. 1.3.

Roadmap In this chapter we address the following questions: Can a consist trend be found within the noisy sample space shown in Fig. 1.3? Can this trend be explained? Can the contradictory holes/balance arguments be resolved? Specifically, why does multiplying the estimate by a factor usually help? And finally, is this result realistic? Namely, is it reflective of the nature of real inaccurate estimates as provided by users?

To answer these questions, we perform a detailed study of what really happens when f grows, both in terms of performance (Sec. 3.2) and in terms of backfilling activity (Sec. 3.3). This leads to the characterization of a heel-and-toe dynamic, which explains the improved performance as resulting from a shift in system behavior towards (a less fair) SJF scheduling (Sec. 3.4). We then

show why this can break down with higher f values (Sec. 3.5), and pinpoint the burstiness of the load as the main cause for this effect (Sec. 3.6). After the impact of increased f on performance is fully understood, we go on to explicitly quantify the performance/fairness tradeoff. We then argue that multiplying estimates is actually a scheduling technique that exercises this tradeoff, as schedulers can multiply the estimates they use, whereas users' behavior is completely different (Sec. 3.8). In fact, multiplying improves performance regardless of whether the values being multiplied are actual runtimes (perfect) or were supplied by users (flawed); it's just that the more accurate the values we are multiplying, the better the resulting performance becomes (Sec. 3.9). Thus, accurate estimates actually do improve performance, and the f-model is simply inadequate. All these findings are based on simulation of the EASY scheduler. Our final contribution in this chapter is therefore showing that the above understandings also apply to other backfill schedulers (Sec. 3.10).

Methodology This chapter presents the results of more than three million simulations:[1] Understanding the impact of various f values on performance required us to artificially increase and decrease the load of the trace files listed in Tab. 2.1 (see Chapter 2 for an explanation of how this is done). Thus, aside from simulating the original loads, all the trace files are simulated under "high" (SDSC's 84%) and "low" (CTC's 56%) load conditions, yielding 10 trace/load pairs. (SDSC and CTC are only evaluated under two load conditions). Sec. 3.6 uses a fifth trace file, adding another 3 pairs. Further, Sec. 3.10 reevaluates all 13 pairs under two additional schedulers (different than EASY), yielding a total of 39 trace/load/scheduler triplets. Each such triplet is evaluated under 401 different f values: $f \in \{0, \frac{1}{10}, \frac{2}{10}, ..., 10\}$, $f \in \{11, 12, ..., 100\}$, $f \in \{110, 120, ..., 1000\}$, and $f \in \{1100, 1200, ..., 10{,}000\}$. Then, when using the random model, each of the 400 positive fs are simulated with 100 different seeds for the random number generator. Lastly, Sec. 3.8 uses a modified version of the random f-model and therefore re-executes all the experiments again. This yields $39 \times 400 \times 100 \times 2 \approx 3{,}000{,}000$ simulations.

Naming Notation We have chosen f's minimal value to be zero, because this seems to be best aligned with the perception that "zero badness" implies perfect accuracy. However, due to the multiplicative nature of this factor, it is often much more convenient to set the minimal value to 1, in which case we use an uppercase notation: $F = f + 1$. With this, the random model uniformly draws an estimate of a job with runtime r from $[r, r \cdot F]$, and the deterministic model simply sets the estimate to $r \cdot F$. Note that in *all* figures where badness is shown along the X-axis, the random F-model is plotted against the deterministic $F/2$-model, such that both have the same mean.

3.2 Performance as a Function of Badness

Statistical Confidence The first step we take in trying to uncover the impact of increased "badness" ($= f$) on performance is to expose the trends underlying the very noisy Fig. 1.3 (page 10). To this end, we note that somewhat surprisingly no previous work has used the f-model in a statistically sound manner, that is, researchers have consistently inferred performance results associated with a given f from a *single* simulation, despite the model's random component. Fig. 3.1 shows that plotting performance in terms of mean ("random") and 90% interval ("90% confidence", from

[1]The product of nearly a year's worth of one Pentium-IV/3GHz/4GB compute time, finished in about 6 days with the help of a 64-CPU cluster.

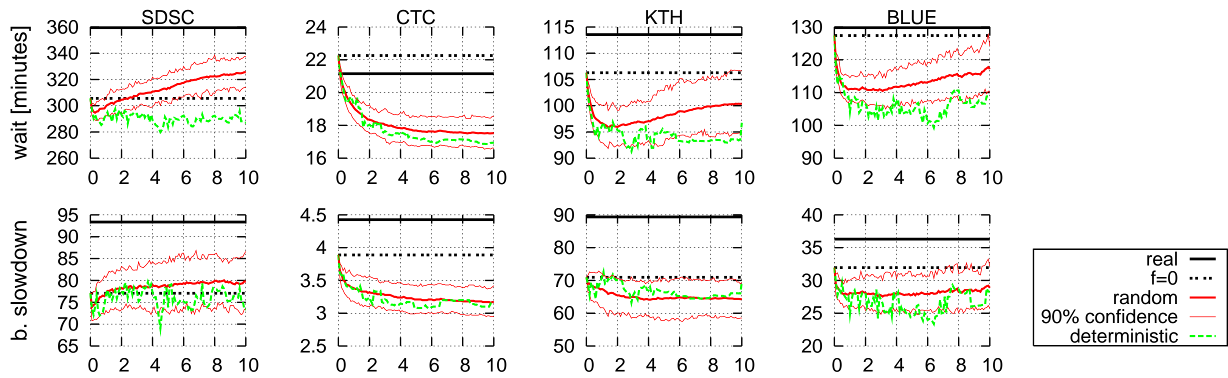


Figure 3.1: *Performance as a function of* $f \in \{0, \frac{1}{10}, \frac{2}{10}, ...\}$*, where each "random" point averages 100 runs with different seeds. The mean results expose clearer performance trends (compare with Fig. 1.3).*

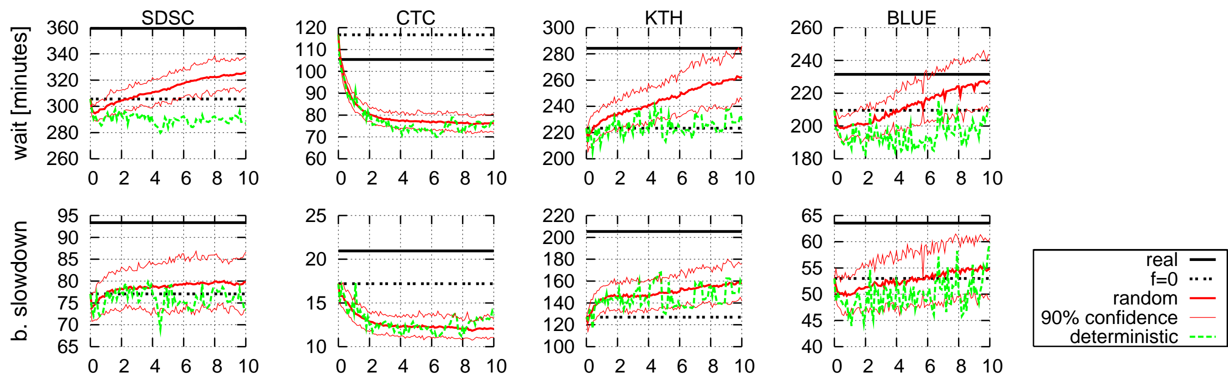


Figure 3.2: *Applying SDSC's high load conditions to KTH and BLUE makes them similar to SDSC.*

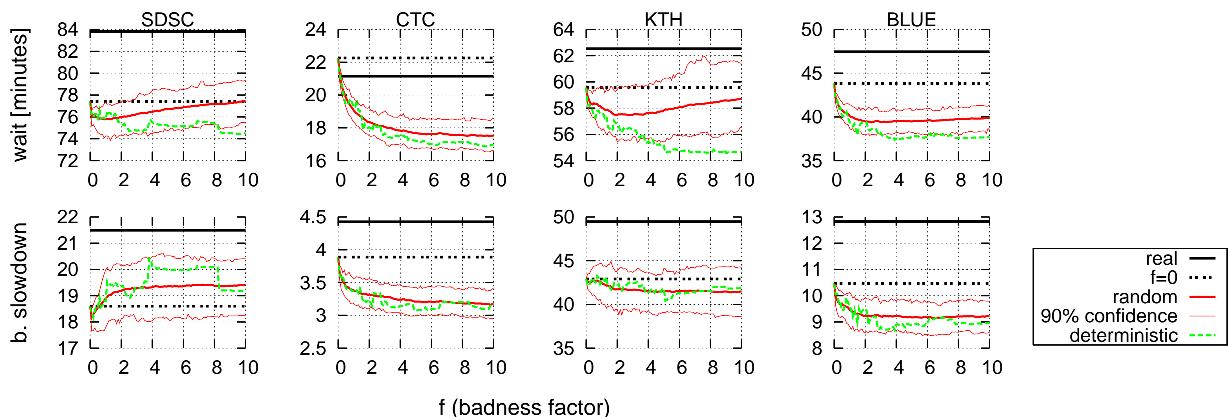


Figure 3.3: *The low load conditions of CTC make the V-curves less pronounced and closer to L.*

the 5th percentile to the 95th percentile) is beneficial, turning the initial noisy results (Fig. 1.3) into relatively smooth curves.

V Trend vs. L Trend Fig. 3.1 reveals two trends: The first is V shaped (most pronounced for SDSC), and the second is L shaped (CTC). In both cases, random performance curves initially

drop (improve) for small f values. Then, the curves either asymptotically converge to some value (L shape), or the trend is first reversed and only then converges (V shape). This general tendency continues to larger f values: BLUE is actually V shaped in both metrics (its curves are quite similar to that of SDSC if changing the X scale to $f \in [0, 100]$ and bigger); KTH/wait and KTH/slowdown are L and V shaped, respectively. The deterministic model obviously stays noisy (only one sample per f), but it is evident that its curves are usually found in the proximity of the lower (better) performance bound of the random model.

Load Correlates with Trends If grouping SDSC and BLUE (V shapes only) and comparing them to CTC and KTH (some L shapes), then Tab. 2.1 (page 32) reveals they can be characterized as having higher and lower load, respectively. To check whether the load determines if curves are V or L shaped, we simulated all the logs under "high" and "low" load conditions. These are chosen to be SDSC's 84% and CTC's 56%, respectively. (Load is varied as explained in Chapter 2). The results are shown in Figs. 3.2-3.3 and suggest that average load is an influencing, yet not the exclusive, factor in determining the performance trend: The trends of SDSC and CTC are invariant to the load change. However, for high load, BLUE and KTH clearly become very similar to SDSC. In contrast, with lower load, the inner-angle of the V curves becomes less "sharp" and somewhat closer to CTC's L curves.

FINDING #3.1

Expressed in terms of confidence intervals, performance is either V or L shaped. Higher or lower average load implies a tendency towards a V or L shape, respectively. The deterministic model is usually closer to the best performance results of the random model.

3.3 Backfilling as a Function of Badness

Holes vs. Balance Our goal is to understand the reason for the system behavior as reported in Finding 3.1. A reasonable first step is to validate or disprove the (contradicting) claims underlying the "holes" and "balance" arguments. Though we already know both fail to provide a full explanation to the observed performance trends (e.g. the V shape), determining which argument (if any) better describes the effect of increased f on backfilling is essential. Recall the holes argument implies backfilling activity intensifies with f, whereas the balance argument claims the effect of bigger holes evens out by backfill candidates appearing proportionally longer.

Results Fig. 3.4 shows the percent of jobs that were backfilled, as a function of f, along with the main characteristics of these jobs. Trends are consistent and confidence intervals are tight. Backfilling rates clearly increase with f. The exact numbers are workload dependent in that higher loads (Tab. 2.1) imply higher rates. But when simulating the logs under equal high/low load conditions (as in Sec. 3.2), the rates become remarkably similar. The runtime/size of backfilled jobs also follow the same pattern, though in this case the increase is invariant to the examined loads.

FINDING #3.2

In accordance to the holes argument and in contrast to the balance argument, bigger fs imply more jobs that enjoy backfilling. On average, these jobs are longer and wider.

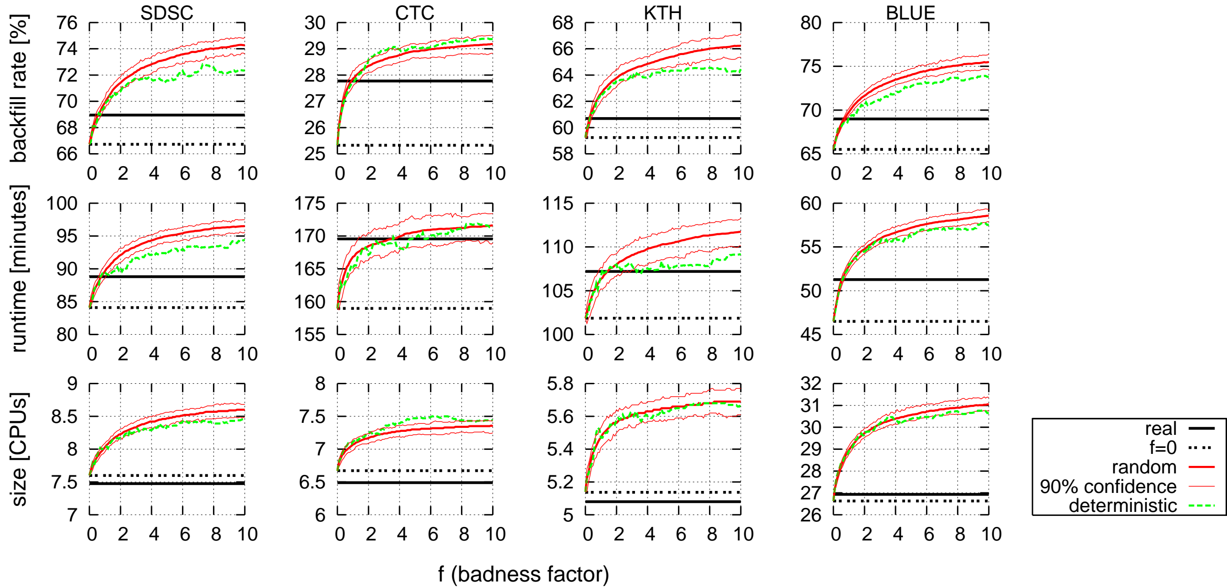


Figure 3.4: *The percent of backfilled jobs and their average runtime and size monotonically increase with f. In all cases, the relative increase is roughly similar, e.g. the rates/runtimes/sizes associated with f=10 are 10-20% bigger than that of f=0.*

This finding can be interpreted as supportive of the L-shaped performance curves (CTC, Fig. 3.1-3.3), based on the notion that jobs can be partitioned into being "light" or "heavy" according to whether their characteristics allow them to be backfilled or not. This interpretation suggests that bigger fs mean more jobs are light and can enjoy better service. However, as we will show below, our finding doesn't just mean *"more"* jobs. It also means *different* jobs, and specifically longer jobs, possibly at the expense of shorter ones.

3.4 The Heel-and-Toe Dynamics

Heel-and-Toe Hypothesis The question that follows Finding 3.2 is why is it so? What's wrong with the balance argument? Why isn't the effect of bigger holes canceled by the backfill candidates that are proportionally longer? After reexamining the backfilling rules, we came up with a possible explanation, as illustrated in Fig. 3.5. To simplify, assume all estimates are exactly double the runtime (F=2 under the deterministic model; recall the uppercase notation defined in Sec. 3.1). Based on the information available to the scheduler at T_0 (time 0), it appears the earliest time for J_3 (job 3) to start is T_{12}, even though the *real* earliest start time is actually T_6. Thus, the scheduler makes a reservation on J_3's behalf for T_{12} and can only backfill jobs that honor this reservation. At T_4, J_2 terminates. As J_1 is still running, nothing has changed with respect to J_3's reservation, and so the scheduler scans the wait queue in search of appropriate candidates for backfilling. J_4 (the first backfill candidate under FCFS) fits the gap between T_4 and the reservation (T_{12}) and it is therefore backfilled, effectively pushing back the real earliest time at which J_3 could have started from T_6 to T_8. Likewise, when J_1 terminates J_5 is backfilled, and when J_4 terminates J_6 is backfilled, pushing J_3's real earliest start time to T_9 and then T_{10}.

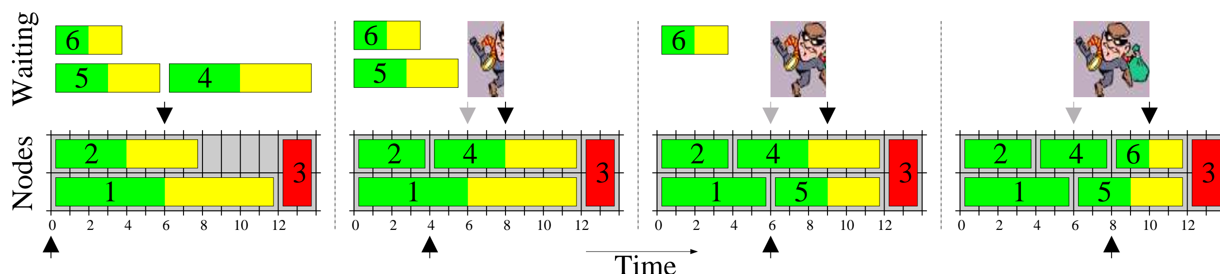


Figure 3.5: *Illustrating heel-and-toe. Job numbers indicate arrival order. Job estimates are exactly double their runtime (F=2). The left portion of jobs (green/dark) indicates their real runtimes. Due to the doubling, the scheduler views jobs as twice as long (right portion; yellow/bright). The bottom arrows show the progress of time, whereas the top black arrows show the earliest time at which job 3 would have been started, had real runtimes been known (at that particular point in time). The thief's width shows the amount of "stolen" time, at the expense of job 3.*

SJFness This "heel-and-toe" scenario, of repeatedly pushing away the earliest starting point of the first queued job, step by step, may continue until T_{12} is reached. During this time, the window between the current time and the reservation time is continuously shortened, such that waiting jobs that fit this open gap get shorter and shorter, effectively nudging the system towards Shortest-Job First (SJF) scheduling. (Note that the initial open gap can be very short to begin with). And so, if the heel-and-toe dynamic does in fact occur, this limited form of "SJFness" contributes to the performance improvement reported in Finding 3.1, namely, the first (descending) part of the V-curves, and the L-curves in their entirety. This effect is directly quantified in the next section.

Tendency towards SJFness with positive f was also observed (but not explained) by Zotkin and Keleher [174], which conducted an "off-line" simulation of what happens when *all* the jobs in a trace arrive at the same exact time instance. They found that, in comparison to f=0, shorter jobs leave the system at a faster rate when estimates are set to be five times the actual runtime.[2] The heel-and-toe dynamics explain this phenomenon.

Verifying Heel-and-Toe Occurs Let J_h be the first queued job (meaning J_h isn't backfilled, but rather, it waits for its turn, becomes first, and gets a reservation). Let S_h denote the *real shadow time* of J_h, defined to be J_h's (hypothetical) start-time, if all estimates suddenly become completely accurate. For example, the initial real shadow of J_3 in Fig. 3.5 is T_6. During the time J_h is first, we say that a backfill operation is *wild* if S_h is pushed away because of it, or that it's *mild*, otherwise. All the backfill operations in Fig. 3.5 are wild, because all resulted in a change of the real shadow. By definition, showing that wild backfilling happens means proving that heel-and-toe dynamics indeed occur. Fortunately, detecting wild backfilling is easy within a simulation: We compute S_h by traversing the run-list in (real) termination order and finding the earliest time in which enough free processor accumulate to satisfy J_h. By doing this before/after a backfill operation, we can tell if the operation is wild (S_h changed) or mild (stayed the same).

Fig. 3.6 clarifies that the heel-and-toe dynamic is not just hypothetical, e.g. with f=10, 2-5% of the jobs are wildly backfilled. The X-axis doesn't start at zero, because there can be no wild backfilling with perfect estimates (the first real shadow is always the last). The consequences of wild backfilling are *delayed* jobs that suffer from at least one wild backfill operation while they are at the head of the queue (as J_3 in Fig. 3.5). Fig. 3.7 (top) shows that around 1% of the jobs are

[2]Though paradoxically this didn't prevent Zotkin and Keleher from using the balance argument [174].

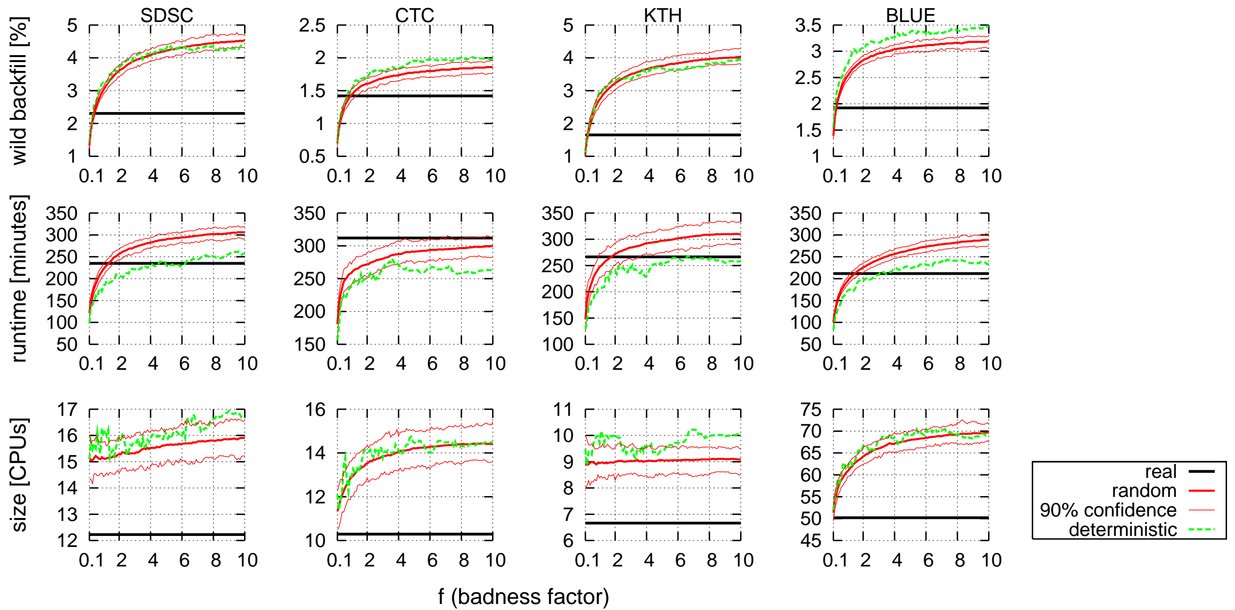


Figure 3.6: *Existence of wild backfilling demonstrates heel-and-toe dynamics occurs. The rate of wild jobs and their average runtime/size follow the same trends as in the general case (Fig. 3.4), but wild jobs are longer and wider. Increasing the load has similar effects to those witnessed in Sec. 3.3 for the general case.*

delayed. Any performance improvement obtained by the f-model is at the expense of these jobs. The average number of times S_h is pushed away is shown in the middle of Fig. 3.7 (three times for J_3 in Fig. 3.5). Finally, the bottom of Fig. 3.7 shows the average delay duration. This is the elapsed time between J_h's initial real shadow and its eventual start time (the "stolen" time in Fig. 3.5).

Holes vs. Balance Revisited Our findings indicate that the seemingly contradictory "balance" and "holes" arguments can in fact be reconciled: The performance improvement attributed to positive fs is not just because of wider holes in the schedule that allow for more backfilling (in accordance to the "holes" argument), because backfill candidates are indeed widened proportionally (in accordance to the "balance argument"). Rather, it is the result of a heel-and-toe effect, which manages to keep the holes open by backfilling shorter jobs that repeatedly delay the execution of the first queued job and lead to an SJF-like schedule.

FINDING #3.3

The heel-and-toe dynamic (1) is verified to occur in practice, (2) reconciles between the balance and holes arguments, and (3) leads to a limited form of SJFness. Thus, it explains the performance improvement due to positive f values.

Let us now explain why performance can also become worse.

3.5 Countering the SJFness of Heel-and-Toe

We now focus on the second, ascending, part of the V-shaped performance curves where performance continuously degrades (Finding 3.1; Fig. 3.1-3.3). The explanation has two components:

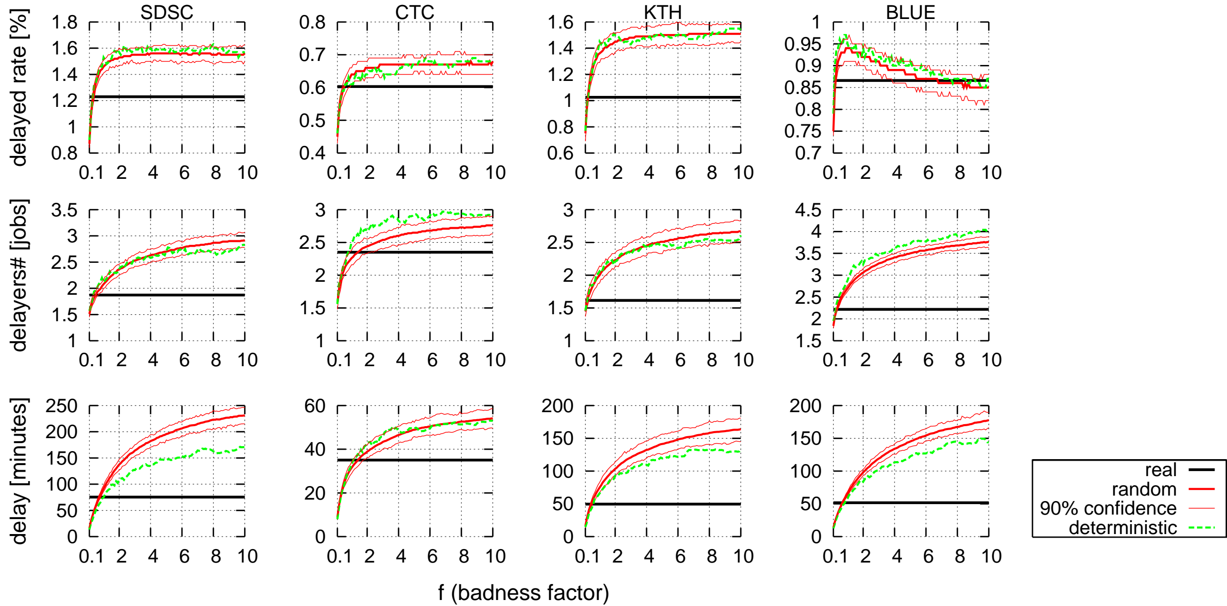


Figure 3.7: *The rate of jobs that suffer from wild backfilling (top), the average number of wild events per such job (middle), and the average delay (bottom). Note that multiplying the top and middle curves results in the top of Fig. 3.6. The unique trend observed in BLUE (top) is also displayed by the other logs if simulating high load conditions as in Sec. 3.2 and examining a slightly wider f range; on the other hand, BLUE becomes like all the others if simulating low load conditions.*

the increased f, and the resulting amplification of randomness (for the non-deterministic model). These components increasingly contradict the SJFness reported earlier:

Increased f As shown in Fig. 3.4 (and highlighted in Finding 3.2), backfilling activity monotonically increases with f, while at the same time, the runtime of backfilled jobs becomes longer. Longer average runtime wouldn't have been problematic by itself, had short jobs been nevertheless prioritized. But this is not the case. To illustrate why, let us reconsider the scenario depicted in Fig. 3.5. Tab. 3.1 lists the estimates of jobs at time T_4 (after J_2 terminates) for various F values, as well as the length of the resulting hole. The last row simply specifies what is shown in Fig. 3.5 (F=2). Recall that job indexes indicate arrival order, used by the scheduler when searching for backfill candidates. Thus, J_4 is the first candidate and since it fits the existing hole it is chosen for backfilling. However, if the value of F had been $1\frac{1}{3}$ instead of 2 (second row in Tab. 3.1), then the hole would have been proportionally smaller and the scheduler would have deemed J_4 as too long for backfilling, favoring instead the shorter J_5 for execution. If F was further reduced to 1 (complete accuracy; first row), than J_5 would also appear as too long, effectively making J_6 (the shortest waiting job) the only eligible candidate. We can therefore see there's a subtle tradeoff here:

FINDING #3.4

> While bigger f means more backfilling (which short jobs enjoy more than longer ones), the bigger holes do in fact allow longer jobs to backfill.

This finding is verified in Fig. 3.8. First, the top row shows the average runtime of non-backfilled jobs: this usually becomes shorter with increased f, suggesting the scheduler indeed

F	estimates				hole length
	J_1	J_4	J_5	J_6	at T_4
1	6	4	3	**2**	2
$1\frac{1}{3}$	8	$5\frac{1}{3}$	**4**	$2\frac{1}{3}$	4
2	12	**8**	6	4	8

Table 3.1: *The length of the hole in the schedule and the estimates of jobs in Fig. 3.5, for various F values. The first row (complete accuracy) lists job runtimes and therefore estimates in later rows can be obtained by multiplying this row with the appropriate F. The hole size is J_1's estimate minus 4 (the current time is T_4, thus 4 time-units have already elapsed). For each F, the estimate of the first job that fits the hole appears in bold.*

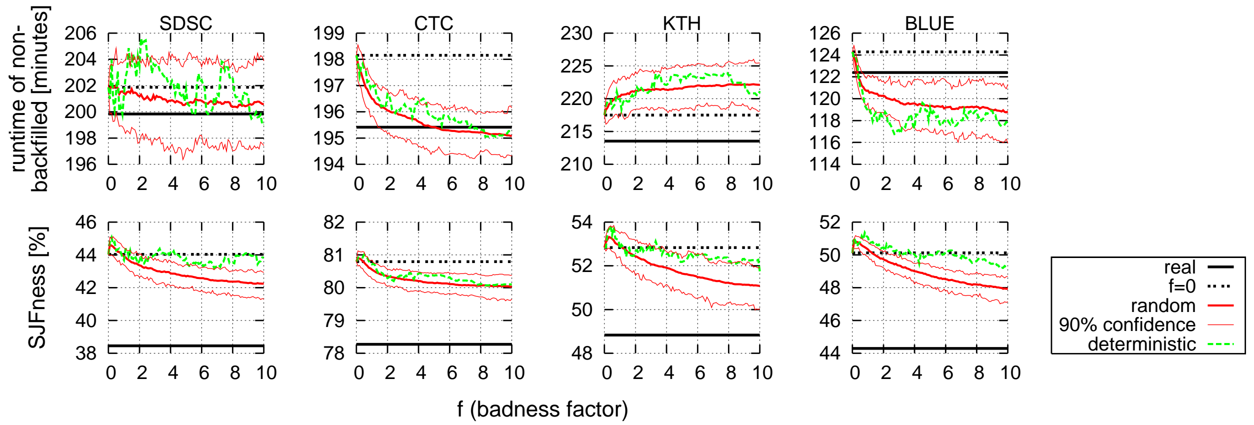


Figure 3.8: *Average runtime of non-backfilled jobs is usually made shorter when increasing f (top). Average SJFness initially rises, but there's a quick trend change as backfilled jobs become longer.*

makes "wrong" decisions by forcing shorter jobs to wait and preferring longer jobs for backfilling (Fig. 3.4). More important is the bottom row that directly quantifies the effect: "SJFness" is the percent of jobs that are the shortest in the waiting queue at the time they are chosen to run. Evidently, SJFness intensifies with very small f values, only to monotonically drop later on (perfectly coinciding with our explanation above).

Increased Randomness The situation gets worse when randomness is introduced, as now, in addition, long jobs can masquerade as short jobs and vice versa. To illustrate this, let J_1/J_2 be two jobs within the wait queue with runtimes r_1/r_2 and estimates e_1/e_2 that were generated by the random model, respectively. This is depicted in Fig. 3.9 (left), assuming $r_1 < r_2$ without loss of generality. We are interested in $Pr\,(e_1 > e_2)$, that is, the probability the scheduler is erroneously told that J_1 is longer than J_2. By conditioning (Bayes' theorem) this is

$$Pr\,(e_1 > e_2) \;=\; Pr\,(e_1 > e_2 \mid e_1, e_2 \in \alpha\,) \cdot Pr\,(\,e_1, e_2 \in \alpha\,) + Pr\,(e_1 > e_2 \mid \overline{e_1, e_2 \in \alpha}\,) \cdot Pr\,(\,\overline{e_1, e_2 \in \alpha}\,)$$

where $\alpha \equiv [r_2, Fr_1]$ is the intersection between the two domains from which e_1 and e_2 are drawn. The second term in the above summation is obviously zero (when either e_1 or e_2 are outside α then $e_1 < e_2$) and so we are left with

$$Pr\,(e_1 > e_2) \;=\; \underbrace{Pr\,(e_1 > e_2 \mid e_1, e_2 \in \alpha)}_{\lambda_1} \cdot \underbrace{Pr\,(e_1 \in \alpha)}_{\lambda_2} \cdot \underbrace{Pr\,(e_2 \in \alpha)}_{\lambda_3}$$

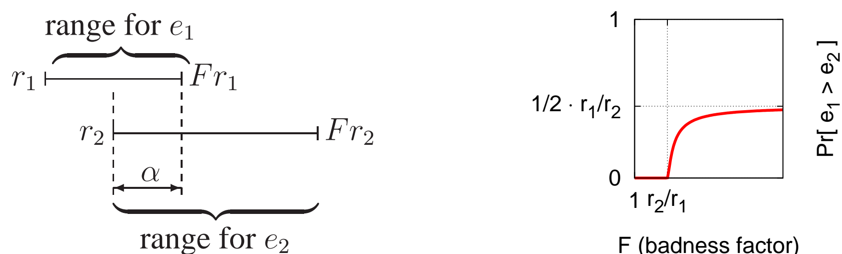


Figure 3.9: *Left: r_i and e_i ($i = 1, 2$) are the runtime and estimate of job J_i, such that e_i is uniformly chosen from $[r_i, Fr_i]$. Right: probability that $e_1 > e_2$ when actually $r_1 < r_2$.*

If α exists ($Fr_1 > r_2$), then $\lambda_1 = \frac{1}{2}$, because it's simply the probability one number is bigger than another if both are uniformly chosen from the same domain. (If α is degenerate then $\lambda_1 = 0$.) As λ_2 and λ_3 represent standard events in a uniform setting, we get

$$Pr\,(e_1 > e_2) \;=\; \frac{1}{2} \cdot \frac{Fr_1 - r_2}{Fr_1 - r_1} \cdot \frac{Fr_1 - r_2}{Fr_2 - r_2} \quad \underset{F \to \infty}{\longrightarrow} \quad \frac{1}{2} \cdot \frac{r_1}{r_2}$$

because the supremum limits of the middle and third factors (in the above multiple) are 1 and $\frac{r_1}{r_2}$, respectively, when F goes to infinity. Thus, the error probability is monotonically increasing, as depicted in Fig. 3.9 (right).

FINDING #3.5

> Under the random model, the bigger the f, the more it is probable the scheduler would erroneously view short jobs as long and vice versa. This explains why SJFness is higher for the deterministic model (Fig 3.8) and hence why the deterministic model consistently outperforms the random model (Fig. 3.1-3.3).

3.6 The Role of Burstiness

CTC is Different We have now managed to explain all the observed performance trends, both the descending and the ascending parts of the curves in Fig. 3.1-3.3. The remaining missing piece in the puzzle is the inherent difference between CTC and the other logs, best observed in Fig. 3.2 that shows performance trends under high load conditions (84% utilization across all logs, as in SDSC). Clearly, the trend of CTC is L-shaped, whereas the others are V-shaped. The question is therefore what makes CTC "immune" to high load conditions? How does it manage to "escape" the destructive processes outlined in Sec. 3.5?

Momentary Load To answer this question, we first define the *momentary load* at time T to be the total number of running/waiting processes (not jobs) that are present in the system at that time instance, divided by the size of the machine. For example, if a machine with 10 CPUs is currently running 8 processes (leaving 2 CPUs idle), while two jobs of size 6 are waiting in the queue, then the momentary load is $(8 + 6 + 6)\,/10 = 2$. The momentary load induced by EASY (with real user estimates) is shown in Fig. 3.10 for when the (offered) load is made equal to that of CTC (top; associated with Fig. 3.3) and that of SDSC (bottom; associated with Fig. 3.2) by means of manipulating arrival-times as explained in Chapter 2. Whether in its original form (top) or after its

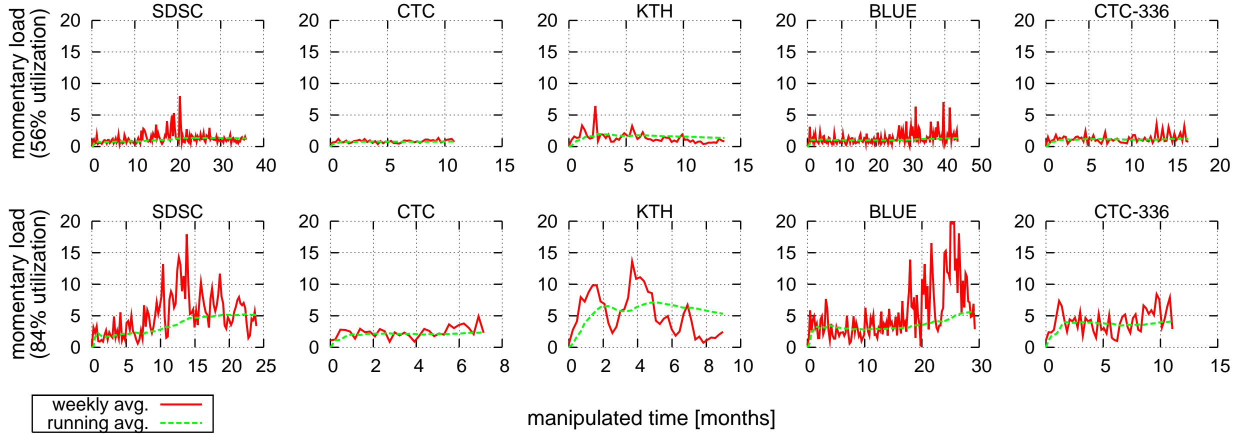


Figure 3.10: *When the overall load is made higher (bottom) by means of arrival-time manipulation, the momentary load exposes a bursty activity pattern in all logs but CTC. Increasing CTC's overall load by manipulating the size of the machine rectifies this (CTC-336).*

offered load has been artificially intensified (bottom), the momentary load of CTC stands out as being "well behaved" and exhibits very little burstiness. We therefore conjecture performance is not just related to the average overall load, but rather, to the manner in which its temporal structure is manifested.

Burstiness Conflicts With SJFness Our conjecture is supported by the fact the effectiveness of the two "anti-SJFness" processes (characterized in Sec. 3.5) is tightly correlated with the size of the wait-queue: To begin with, both processes only apply to jobs that simultaneously populate the queue, dealing with situations where the scheduler considers *currently* waiting jobs for backfilling, and chooses the longer one. A smaller number of concurrently waiting jobs implies such occurrences are less frequent. (E.g., at the extreme, there are one or no waiting jobs, so no scheduling "mistakes" can be made.) Further, the error probability depicted in Fig 3.9 ($Pr(e_1 > e_2) \rightarrow \frac{1}{2}\frac{r_1}{r_2}$) is actually quite small if r_1 is considerably smaller than r_2. But this relates to only one pair of jobs; a crowded wait queue means many pairs are compared, increasing the error-probability proportionally to the wait-queue size. Similarly, the scenarios outlined in Tab. 3.1 only have meaning if $J_4/J_5/J_6$ are ever simultaneously present in the wait queue.

Introducing Burstiness to CTC It is therefore possible CTC's performance trends are qualitatively different because it lacks burstiness. To verify this, we decided to try and raise CTC's offered load (to be equal to that of SDSC) in a manner that will encourage burstiness. The de-facto standard methodology for varying the load of job-scheduling related workloads is with arrival-time manipulation (Chapter 2). However, this is not the only way: Following the intuition illustrated in Fig. 3.11, we've decided to change the load by reducing the size of the CTC machine (512 processors, originally). Luckily this is possible because the maximal job size within the CTC workload is 336. Incidentally, changing the machine size from 512 to 336 processors yielded a workload with 85% utilization, very close to the desired target load of 84%. We call this modified workload CTC-336, and present its momentary load at the right of Fig. 3.10 (CTC-336's 85% was made equal to SDSC's 84% and CTC's 56% by standard arrival-time manipulation). Our attempt is indeed successful, as CTC-336 is clearly more bursty than CTC. Indeed, examining the performance trends of CTC-336 in Fig. 3.12 reveals our hypothesis was right:

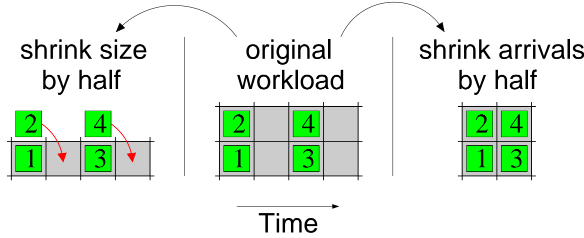


Figure 3.11: *Increasing the load of a given trace (middle) by arrival time manipulation (right) or by reducing the size of the machine (left). Left displays a burstier pattern (2/1/2/1) relative to the right (1/1), though both have 100% utilization.*

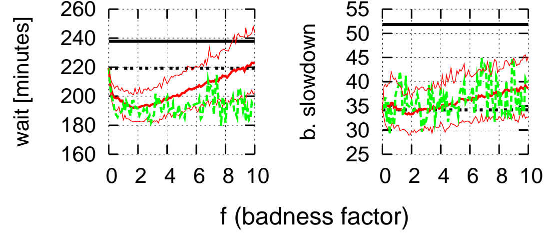


Figure 3.12: *Performance of CTC-336 (high load conditions; compare with Fig. 3.2). In contrast to the original CTC, which was associated with L performance curves, CTC-336 is V-shaped.*

FINDING #3.6

Performance trends tend to be either V- or L-shaped, depending on whether the workload is bursty or not, respectively.

3.7 Unfairness as a Function of Badness

The heel and toe dynamics suggest that the performance improvement obtained by multiplying is at the expense of jobs that get a reservation, which are usually both long and wide. (In the four logs their average runtime, estimate, and size are approximately 4h, 7h, and 17% of the machine's processors, respectively.) It therefore appears as if increasing f is "unfair". However, to directly verify that this is true, we need a way to measure fairness. Such a metric is described next.

Given a job J, assume it is possible to calculate the hypothetical time in which it is "most fair" to start this job. This is called J's *fair start time*, denoted $FST(J)$. Let the *actual start time* of J (under the scheduler we happen to evaluate, which is EASY in our case) be denoted as $AST(J)$. Using this notation, Sabin and Sadayappan defined the average *unfairness* as

$$\frac{1}{|jobs|} \sum_{J \in jobs} \max\left(0, AST(J) - FST(J)\right)$$

where the $jobs$ set contains all the participating jobs [123]. Note that this metric expresses time (e.g. minutes), which is the per-job average delay period beyond what is "most fair". Also note that the term involving $\max$ insures the summation includes only nonnegative values, and that only jobs which were treated unfairly contribute positive quantities. (These may be "delayed jobs" in our terminology from above.)

The remaining missing piece is how to compute $FST(J)$. This was defined by Srinivasan et al. [141] as the start time of J under a hypothetical "conservative" backfill scheduler (assigns reservations to *all* waiting jobs; see Sec. 1.1.2) that has two unique properties: (1) it utilizes completely

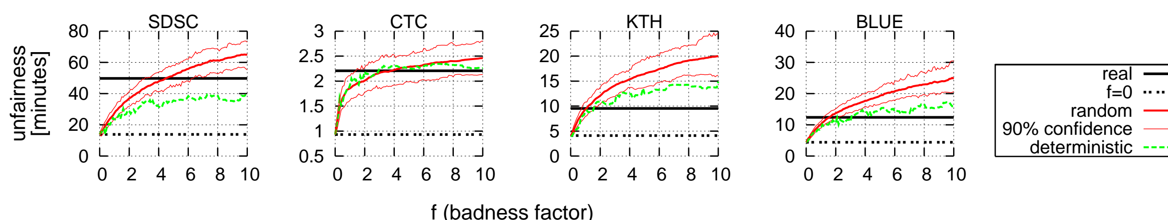


Figure 3.13: *Unfairness as a function of badness. Increased badness translates to a less fair schedule.*

accurate estimates, and (2) it suddenly changes the scheduling strategy to strict no-backfill FCFS at the exact time instance in which J arrives.[3]

An attractive property of this definition is that it is at least as fair as using a strict FCFS definition (the order of the FSTs is in perfect alignment with the order of job arrivals and no job is ever delayed due to later arriving jobs), but it is nevertheless "efficient" enough to be useful, allowing for a meaningful evaluation when judging the fairness of high-end schedulers. (Unfortunately, in comparison to FCFS start times, virtually all high-end schedulers may have zero unfairness, due to the poor performance of FCFS).

The results are shown in Fig. 3.13 and all have a Γ-like shape indicating that increased f indeed yields increased unfairness, a fact that perfectly coincides with the heel and toe description. Note that, with f=0, unfairness is smaller than when real estimates are employed. However, as f is increased the situation is quickly reversed. The conclusion is therefore that

FINDING #3.7

> Multiplying all estimates by a factor is actually *trading off fairness for performance*.

3.8 Making the Model More Realistic

The Problem The f-model is *the* dominant model for generating artificial user runtime estimates. It is used to complement workloads that lack estimates data [169, 56, 58], but more importantly, to evaluate the impact of inaccurate user estimates on backfilling algorithms [146, 47, 174, 169, 108, 15, 142, 170, 122, 34, 64]. Based on the f-model, researchers have drawn neat conclusions that range from "performance is independent of accuracy", through "what the scheduler don't know won't hurt it", to "inaccuracy actually improves performance" (see Sec. 1.2.1, page 9). Indeed, when employing *artificial* estimates as generated by the f-model, these claims may reflect certain aspects of the truth, as shown above. However, there is a fundamental, yet evidently very elusive and overlooked, problem with all the insights that are based on the f-model:

THE PROBLEM WITH THE f-MODEL

> Increased inaccuracy that is modeled by greater f values effectively *spreads* the estimates across a larger domain. But with real estimates it's exactly the opposite! Namely, inaccuracy manifests itself by more jobs using the *same* estimate value. Thus, conclusions based on the theoretical f-model might not apply when real user estimates are involved.

[3] It's as if we're using a different scheduler for each job; however, with the right data structure, FSTs may actually be computed during one sequential $O(n)$ pass through all the jobs, requiring no simulation.

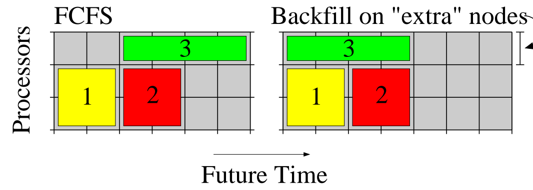


Figure 3.14: *FCFS cannot start J_3 before J_2. But with backfilling, if more nodes than needed will be available for J_2 at its reservation time, these "extra" nodes can be allocated immediately.*

Understanding results that are based on the f-model can be interesting and important. For example, the heel-and-toe dynamics turned out to be the reason why, as shown in Fig. 1.2, doubling of *real* user estimates improves performance. (Doubling is a legitimate scheduling optimization as will be discussed in Sec. 3.9.) Nevertheless, such understandings can have only limited applicability for real systems that employ real user estimates. Importantly, a statement like "inaccuracy improves performance" is a misleading oversimplification: real inaccuracy is actually tightly correlated with degraded performance, as will be exemplified next.

Modality Human users do not choose estimates that are uniformly distributed between the real runtime and its multiple with some value, but rather repeatedly use the same "round" estimates (5 minutes, 1 hour etc). Indeed, we find that 90% of the jobs use the same 20 "round" values (see Chapter 5), a fact that explains the staircase-like CDF curves shown in Fig. 1.14 (page 22). As noted in the associated table, a value that always enjoys immense popularity is E_{max} (the maximal estimate allowed), used by 10-27% of the jobs, which typically makes it the most popular estimate. This is probably due to a combination of users lacking the ability to provide good estimates, along with the strict policy of backfill schedulers to kill underestimated jobs. (Nevertheless, even when this policy is not enforced, the improvement in the quality of user estimates is apparently negligible [93], which means the former argument is most likely more detrimental.)

E_{max}'s popularity has dire implications on performance. To understand why, consider an extreme case in which *all* jobs use E_{max} as their estimate. We claim that in such a case, backfilling activity (as shown in Fig. 1.1, page 5) completely stops. The proof's outline is the following. The reservation of the first queued job is computed based on estimated termination times of currently running jobs, and these will all occur before E_{max} time, by definition. Hence, the reservation itself will occur before E_{max} time, and therefore backfilling holes (from the present time until the reservation) are always smaller than E_{max}. Since we assume all estimates of waiting jobs are exactly E_{max}, we get that none will fit the holes in the schedule. The consequence is that scheduling largely reverts to plain FCFS, resulting in a serious performance degradation. (The only remaining backfill activity is on the expense of the "extra" nodes [108], as shown in Fig. 3.14.)

Previous studies have neglected to take E_{max} into account. For example, it has been conjectured that the connection between longer execution time and better accuracy shown in Fig. 1.15 (page 22) is because the more a job progresses in its computation, the grater its chances become to reach successful completion [20]. But the reason is actually much more prosaic: since (1) E_{max} is an upper bound on estimates, and (2) backfilling insures estimates are bigger than runtimes, we have

$$runtime \leq estimate \leq E_{max}$$

Thus, as runtimes get bigger (closer to E_{max}), the accuracy fraction ($= \frac{runtime}{estimate}$) converges to 1. Further, the various peaks in Fig. 1.15 are due to other popular estimates (smaller than E_{max})

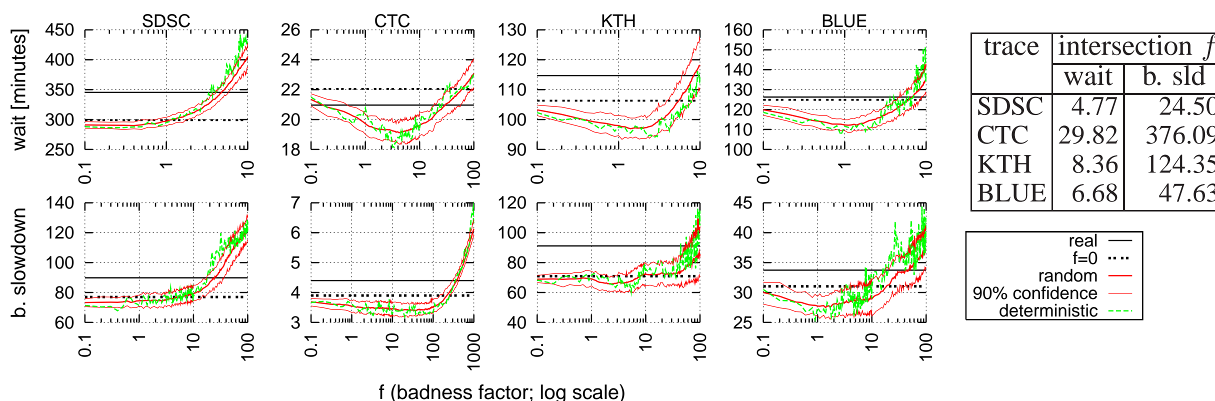
and the many *underestimated* jobs that used them: as these jobs are killed upon reaching their estimates, they have 100% accuracy. But many other jobs that use these popular values are in fact significantly *overestimated*. The problem is that the scheduler has no way to distinguish between such jobs, in contrast to when the f-model is used. To clarify, consider a scheduler that explicitly favors shorter jobs for backfilling [174, 15, 156] and must work with inaccurate estimates. If these estimates nevertheless result in a relatively correct ordering of waiting jobs (as would happen with the f-model), performance can dramatically improve (up to an order of magnitude according to [15]). However, if estimates are modal (as generated by real users), many jobs look the same in the eyes of the scheduler, which consequently fails to prioritize them correctly, which means performance deteriorates. As shown earlier, heel-and-toe dynamics nudge a FCFS-based scheduler towards SJFness, and therefore the same argument applies. Further, an estimate distribution that is dominated by only a few monolithic modes (E_{max} and others) negatively effects performance, because less variance among waiting jobs means less opportunities for the scheduler to exploit existing holes (with various sizes) for backfilling.

Enforcing an Upper Bound on Estimates The bottom line is that if one wants to model increasing user inaccuracy, one should focus on the modality of user estimates. For example, 10% of the jobs using E_{max} is an optimistic scenario relative to 20%, which in turn is more optimistic than 30%, etc. Modeling increased inaccuracy by gradually associating more jobs with E_{max} is certainly more realistic than using the vanilla f-model. Fortunately, E_{max} can be easily incorporated within the f-model if instead of using artificial estimates as is, we truncate them to be E_{max} in case they are bigger. Namely, if the artificial estimate is e, we instead use $\min(e, E_{max})$. Let this be denoted as the *truncated f-model*. This model has the property that bigger f values imply more jobs associated with E_{max}.

Fig. 3.15 shows the results. The truncation has negligible impact for very small f values, because at this points very few artificial estimates exceed E_{max}. The common trend is therefore of improved performance, similarly to the vanilla f-model. Truncation gradually becomes the dominant factor as f increases and so the trend is reversed. The difference between the truncated (Fig. 3.15) and vanilla (Fig. 3.1) models when f goes to infinity is that the ascending part of the latter never[4] intersects the curves associated with real user estimates (verified till f=10,000), whereas the former always does. At the intersection point, the truncated model is successful in "capturing the badness" of the real estimates. Thus, with big enough f, the behavior of the truncated model coincides with our claim above that performance degrades if inaccuracy is increased by making the estimate distribution more modal.

An Accurate Model While the truncated f-model is more realistic than the vanilla one, its output is still fundamentally different from the real thing. A key difference is that only one mode is created (at E_{max}), whereas real estimates exhibit several modes (Fig. 1.14, page 22). Indeed, the E_{max} mode is the most influential, but other modes are also essential in that they significantly contribute to the overall observed effect of bad performance in the face of real user estimates. Further, the E_{max} mode as created by the model is poorly constructed: it consists of long jobs only (with big enough runtimes such that multiplying them with F results in estimates bigger than E_{max}). In reality, many short jobs are estimated by users to run E_{max}. Of these, most notable are jobs that fail on startup. Thus, even with the truncated model, the scheduler can still identify shorter jobs better than when real estimates are employed (until a certain F).

[4]With the exception of BLUE/wait.



trace	intersection f	
	wait	b. sld
SDSC	4.77	24.50
CTC	29.82	376.09
KTH	8.36	124.35
BLUE	6.68	47.63

Figure 3.15: *Performance results obtained with the truncated f-model (compare with Fig. 3.1). The table specifies the intersection point between curves associated with the "random" model and those associated with real estimates. (Slight differences exist between results associated with real user estimate of the vanilla and the truncated models. This is due to runtimes bigger than E_{max} that unexplainably exist in the original logs and were truncated to make sure they are not bigger than the associated estimates.)*

For these reasons, we find ourselves in an undesirable situation where each trace/metric combination requires a different f to obtain performance results comparable to that of the real thing (table at right of Fig. 3.15). This serious drawback is contrasted with the model's simplicity and ease of implementation and use. We therefore view it as the "quick and dirty" substitute for the vanilla version, namely, if faced with the choice of using either one of them, we strongly support the truncated version. It is our opinion that while it is not perfect, it is also not "garbage".

In general, however, we advocate using the more sophisticated estimate model developed in Chapter 5, instead of the f-model variants. This chapter serves in part as motivation. The input of our new model is E_{max} and optionally the percent of associated jobs.[5] The optional argument allows to gradually increase inaccuracy in a truly realistic manner. The output of the new model is a series of modes, where each mode is a pair consisting of an estimate value and the percent of jobs that use it (twenty of which cover 90% of the jobs). This means that in contrast to common practices, estimates are not generated on a per-job basis, but rather, collectively, before hand. Thus, our model also provides a way to map the generated distribution onto a set of jobs with predetermined runtimes, such that each job's assigned estimate is equal to or bigger than its runtime, as required by the backfilling rules. The model is available for download at [155], and was verified to produce results that are almost identical to the real thing [157] (Chapter 5).

3.9 Practical Implications

Our results so far that were obtained under the f-model have mostly a theoretical value, because we are multiplying completely accurate runtimes and this information is normally not available a-priori to the scheduler. Nevertheless, the results do have practical implications, both in terms of system design and implementation, and in terms of system evaluation.

It turns out that our understandings regarding the act of multiplying *prefect* estimates (= run-

[5] We show that the dissimilarity between estimate distributions of different traces is largely embodied in the percent of jobs that use E_{max} as their estimate; the distributions are otherwise remarkably similar.

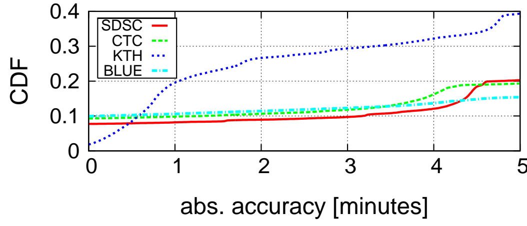


Figure 3.16: *CDF of absolute accuracy (difference between estimate and actual runtime of jobs).*

SDSC	CTC	KTH	BLUE
0.47	0.49	0.75	0.58

Figure 3.17: *Correlation between runtimes and estimates.*

times) also hold when multiplying *real* estimates, as were given by users. Intuitively, this is so because a non-negligible portion of the jobs reach their estimates and are killed by the system, which leads to 100% accuracy for such jobs (peaks in Fig. 1.15, page 22). Further, despite the popularity of E_{max}, most estimates are nonetheless rather short (Fig. 1.14, page 22), and since estimates serve as runtime upper bounds then jobs with short estimates are guaranteed to indeed be short. In this respect, *relative* accuracy $\frac{r}{e}$, can be less important than *absolute* accuracy $e - r$ (actual time difference between an estimate and the associated runtime). Fig. 3.16 shows the CDF of absolute accuracy, which is zero for 2-10% of the jobs, less than one minute for 8-20%, and less than five minutes for 15-40%. Tab. 3.17 summarizes the correlation between runtimes and estimates, which is indeed non-negligible. The bottom line is that user estimates are, to some extent, similar to runtimes. It is therefore not far fetched to expect that the results of multiplying them would be similar too.

Using X_{real} and X_{perf} to denote the cases when multiplying real and perfect estimates, respectively, let us now compare between the two. (X_{perf} relates to the deterministic f-model.) Fig. 3.18 shows most of the backfilling related metrics that we have used so far, as a function of f, for the two alternatives (note that the X-axis is logarithmically scaled and spans 0–10000). Indeed, in all cases the trends are clearly qualitatively similar, albeit have quantitative differences, to be discussed next.

Better Estimates Yield Favoring of Shorter Jobs Specifically, in Fig. 3.18a we can see that for small f values (0–10), there is less backfilling activity with X_{perf}, though it rapidly becomes similar to that of X_{real} for bigger fs. Runtimes of backfilled jobs behave similarly (Fig. 3.18b), but the initial f domain where X_{perf} backfilled jobs are shorter stretches beyond f=10. The situation is different when considering wild backfilling (Figs. 3.18c-d), as there is a difference across the entire f range: the X_{perf} wild jobs are shorter, their rate grows faster and is asymptotically bigger. This means more and smaller heel-and-toe "steps", implying a temporal preference of X_{perf} to backfill shorter jobs sooner. In other words, even though the rate and average runtime of all backfilled jobs (wild+mild) is usually asymptotically comparable for X_{real} and X_{perf} (Figs. 3.18a-b), with X_{perf} shorter jobs wait less before being backfilled. Indeed, by Fig. 3.18e, X_{perf} SJFness is clearly higher, and by Figs. 3.18f-g, this improved SJFness is not at the expanse of the "delayed" jobs, as their rate and delay can be similar, higher, or lower for X_{real} or X_{perf}, depending on the trace.

Better Estimates Yield Increased Fairness Note that even though there can be more delayed jobs with X_{perf} (Fig. 3.18f; CTC/KTH), the delay can be longer (Fig. 3.18g; SDSC/BLUE), and there are certainly more wild jobs (Fig. 3.18c; all logs), the schedule with perfect estimates is nevertheless distinctively more fair (Fig. 3.18h). This is so because the "delay" related metrics are computed with respect to the single reservation EASY makes, which is based on inaccurate

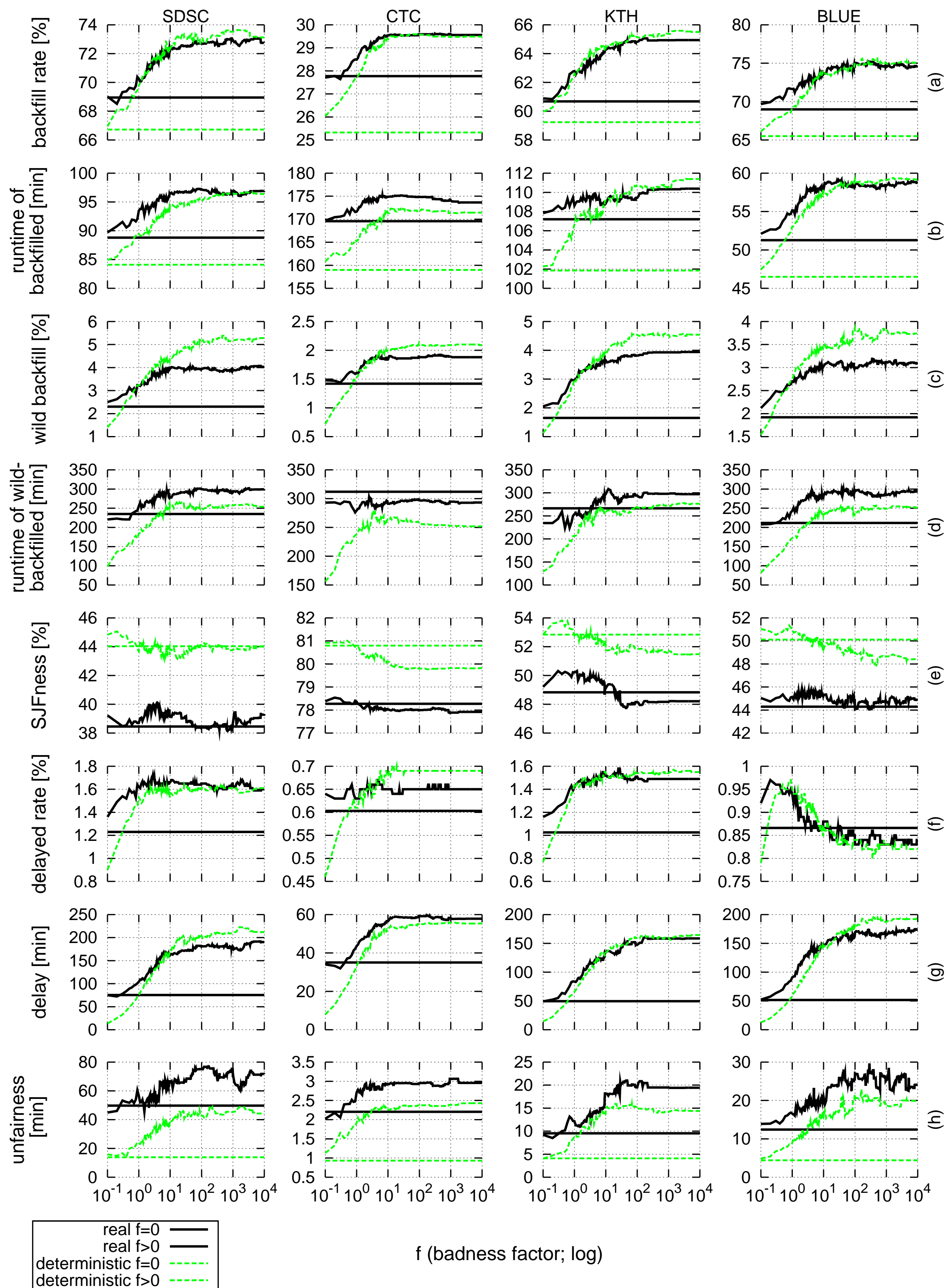


Figure 3.18: *Comparing the impact of multiplying real and perfect estimates on various backfilling aspects.*

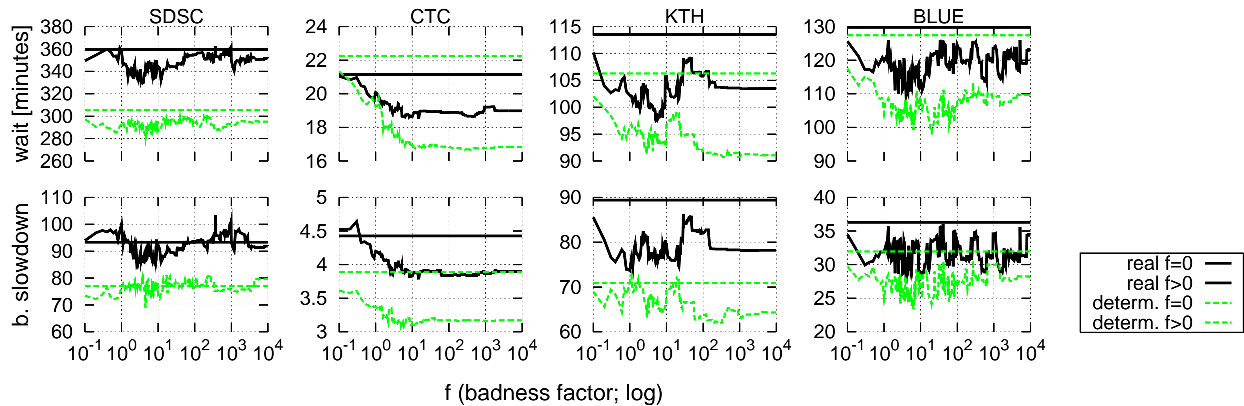


Figure 3.19: *Comparing the impact of multiplying real and perfect estimates on performance metrics.*

estimates. In contrast, the unfairness metric quantifies fairness in absolute terms, relative to a theoretical schedule with perfect information in which all jobs get a reservation and are therefore guaranteed never to be delayed due to any later arriving jobs. Consequently, a job may very well be considered as *not* being delayed, while at the same time be treated unfairly. When taking all this "unfairness" into account, X_{perf} is clearly more fair than X_{real}.

Better Estimates Yield Better Performance The performance results are shown in Fig. 3.19. They are rather noisy due to the inherent noisy nature of the deterministic models. Nevertheless, there are two immediate observations that can be made (these are probably the more important findings of this chapter). The first is that, like with perfect estimates, making real estimates less accurate by multiplying them with a factor $F > 1$ usually improves performance. In combination with Fig. 3.18h ("unfairness") this reinforces Finding 3.7 that multiplying estimates by a factor means trading off fairness for performance.

The second observation is that in contrast to common belief, better accuracy does in fact improve performance in the sense that the more accurate the initial (to be multiplied) estimates are, the better the resulting performance becomes. As seen earlier, in no way does the act of multiplying emulate the inaccuracy exhibited by real users. Rather, it simply adds a certain "SJFness" to the schedule through heel and toe dynamics. Consequently, multiplying is, and should be viewed as, not more than a scheduling "optimization". Indeed, it is both legitimate and practical to configure the scheduler to boost performance at the expense of fairness by means of multiplying estimates. Thus, we contend that artificial inaccuracy (multiplying) is a *property of the scheduler*, whereas "real" inaccuracy is a property of users. The latter manifests itself completely differently and likewise has completely different consequences in terms of performance (Sec. 3.8). The problem is that up till now researchers confused between the two types of inaccuracies. This central argument is summarized in Tab. 3.2 and its bottom line is that

FINDING #3.8

The popular statement that "increased inaccuracy improves performance" is a *false misconception*, originating from a confusion between a scheduling strategy and the nature of users. The correct statement is that increased inaccuracy *worsen* performance, but that the scheduler can boost it at the expense of fairness by multiplying the estimates with some factor.

#	*source of inaccuracy*	*property of*	*nature of inaccuracy*	*effect on performance*	*effect on fairness*
1	real	users	modal & favors E_{max}	worsened	worsened
2	artificial (f)	scheduler	promotes heel & toe	improved	worsened

Table 3.2: *Comparing real and artificial sources of inaccuracy.*

Conclusion The implication of our findings on the design and implementation of systems is exposing the performance-fairness tradeoff that may now be judicially exploited by system designers. The implication of our findings on systems evaluation is revealing that analyses which evaluated the impact of inaccurate estimates on performance were methodologically erroneous if they relied on the f-model, as multiplying by a factor is (1) actually a scheduling strategy that is (2) anything but representative of actual users. A correct evaluation should be done as specified in Sec. 3.8.

3.10 Non-FCFS Backfilling

The results presented in this chapter were all obtained under the EASY scheduler, which is the most popular supercomputer default setting to date [37]. Nonetheless, many other backfilling configurations were proposed and evaluated (see Sec. 1.1.2), and accordingly, contemporary schedulers offer a wide range of tunable policies. The question is therefore whether the results presented here are applicable to other scheduling schemes. In this context, in relation to the initial observation by Mu'alem and Feitelson that multiplying helps [47, 108], Lee and Snavely argued that

MUST-REPROVE CLAIM

"The key point 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" [94].

We contend that the situation is not so bleak and that a generalization is possible.

SJBF Let us first consider the question of what will happen if the scheduler is modified such that traversing the wait-queue in search of the next job to backfill is done in SJF order, instead of FCFS. (We introduce this scheduler in Chapter 4 and name it *SJBF*: Shortest-Job Backfilled First [156]). It would seem that such a scheduling scheme of explicitly favoring shorter jobs will invalidate the heel-and-toe rationale of doing it implicitly. One might therefore expect SJBF performance to be independent of f. Nevertheless, as shown in Fig. 3.20, the results are qualitatively rather similar to that of plain EASY (compare with Fig. 3.19): A positive badness factor usually yields improved performance for the real and deterministic models, even though slowdown is less sensitive to f under the latter. Likewise, the random model yields the familiar V and L shapes we have previously encountered (compare with Fig. 3.1). The reason for this qualitative similarity is that the heel-and-toe dynamics occur even under SJBF. This is true because SJBF reservations are still allocated in FCFS order and therefore the same exact mechanism of repeatedly delaying the first queued job applies. In quantitative terms, SJBF is unsurprisingly better than EASY, because of the increased preference of shorter jobs.

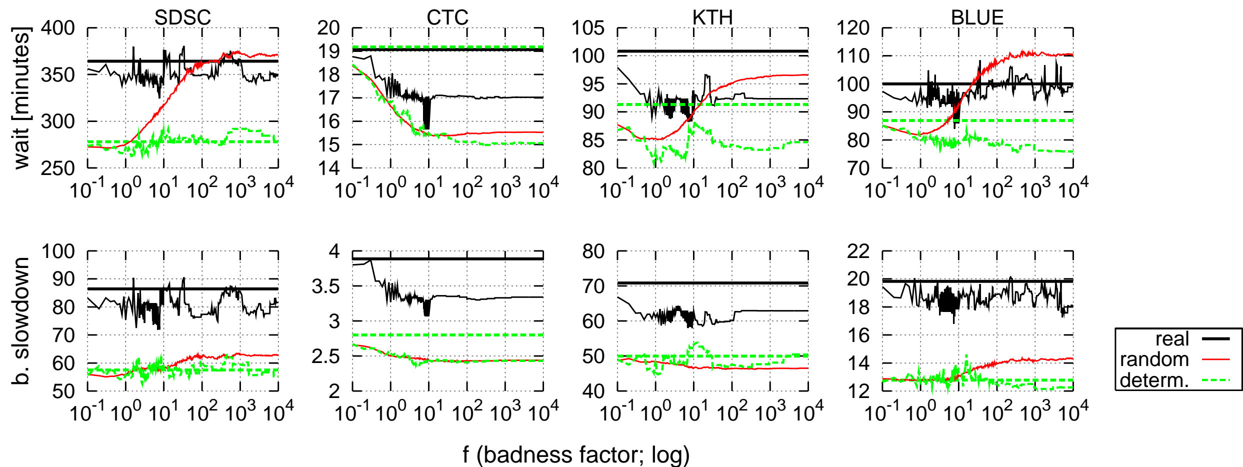


Figure 3.20: *Performance of the SJBF scheduler, as a function of f. As usual, the straight horizontal lines show the performance when using the exact corresponding values, without multiplying them.*

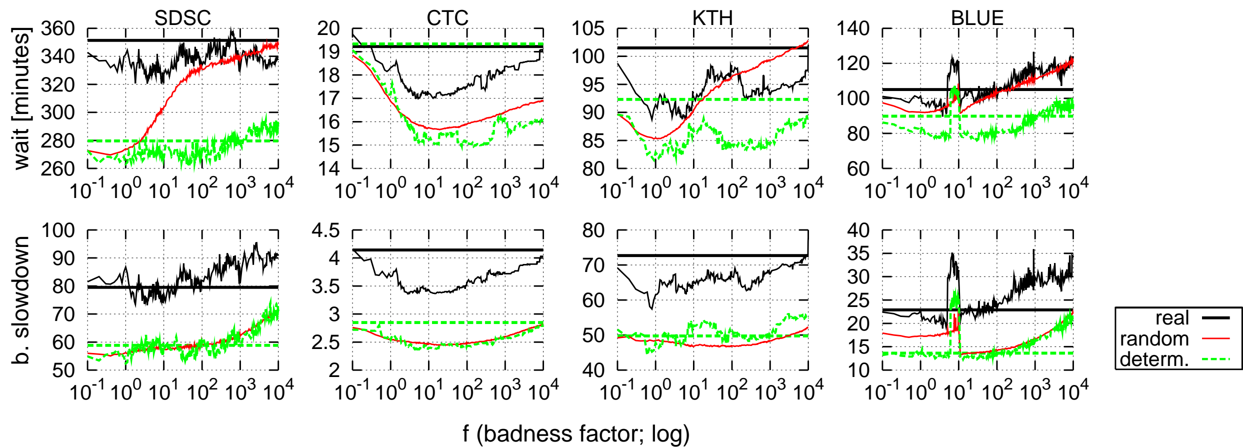


Figure 3.21: *Performance of the LXF&W scheduler, as a function of f.*

LXF&W Another non-FCFS scheduling variant for improving performance as well as fairness is *LXF&W* (Largest eXpansion Factor and Wait time), proposed by Chiang and Vernon [19]. Under this discipline, the priority of a job is given by $\frac{w+e}{e} + 0.02w$, where e is the job's runtime estimate and w is its current wait time.[6] The left term is the estimated "expansion factor", a.k.a. slowdown. This component makes sure that shorter jobs are initially favored, due to having the estimate as the denominator. However, in the interest of fairness, it grows proportionally to w as time elapses, a trend that is further intensified by the addition of $0.02w$. Fig. 3.21 shows the performance results of LXF&W. Surprisingly, instead of being L-shaped, the "real" and "deterministic" curves are U-shaped and usually intersect the associated straight line corresponding to f=0 (somewhat similarly to the results obtained with the truncated model shown in Fig. 3.15). The solution to this mystery lies in the definition of the LXF&W priority function, which when using e.g. the deterministic f-model is actually

[6]Note that (1) both e and w are expressed in hours, which is significant due to the inclusion of $0.02w$ in the LXF&W priority, and (2) due to the w component, the priority is *dynamic* in that it changes over time (recomputed for each scheduling decision).

$$\frac{w+e}{e} + 0.02w \;=\; 1 + \frac{w}{e} + 0.02w \;=\; 1 + \frac{w}{r \cdot f} + 0.02w$$

where r denotes the runtime. When f goes to infinity, this priority obviously becomes dominated by the rightmost term, which means LXF&W converges to plain EASY, as ordering the queue by wait- or arrival-time is completely equivalent. The U shape is therefore explained as follows: The initial performance improvement (left side of the U) is as usual the result of heel-and-toe dynamics, as similarly to other backfilling algorithms, jobs that wait and get a reservation under LXF&W are too both long and wide.[7] At the same time, for smaller fs, LXF&W is significantly better than EASY in terms of absolute numbers, due to its explicit preference of shorter jobs (compare Fig. 3.21 to 3.19, for example, LXF&W's f=0/KTH/deterministic/slowdown is 50, whereas EASY's is 71). Consequently, as f increases, the convergence of LXF&W to EASY becomes the dominant effect, rapidly overshadowing the initial dominance of heel-and-toe dynamics while the significant performance gap between the two algorithms gradually closes. In other words, LXF&W is positioned somewhere between LXF and EASY, such that with smaller fs it is closer to the former and with bigger fs to the latter. Other than that, our findings so far still apply.

Generalizing The bottom line is that the three representatives of the backfilling algorithms class (EASY, SJBF, and LXF&W) have reacted similarly when subjected to estimates that were multiplied by a factor. Based on this observation and on our current understanding of the dynamics of backfilling, we conclude that

FINDING #3.9

As long as reservations are allocated *to promote fairness*, then multiplying of estimates will result in a heel-and-toe effect whereby shorter jobs exploit the wider scheduling holes for backfilling, at the expense of longer/wider jobs. This is completely orthogonal to the specific ordering in which backfilling activity is conducted or reservations are allocated.

Only if reservations are allocated in a way that is unrelated to FCFS fairness (e.g. in SJF order) or are categorically eliminated (e.g. pure SJF) can we expect the impact of multiplying to be different.

3.11 Conclusions

For the conclusion of this chapter, we refer the reader to Section 7.1 (page 122).

[7] Their average runtime, (real) user estimate, and size are 4 hours, 7.5 hours, and 17% of the machine's size, respectively. Thus, multiplying all estimates by a factor helps delay such job in favor of shorter ones.

Chapter 4

Backfilling With System-Generated Predictions Rather Than User Runtime Estimates

4.1 Introduction

Context This chapter was fully introduced in Sec. 1.2.3 (page 13) and Sec. 1.3.3 (page 24), which also conducted a detailed survey of related work. Briefly, recall that backfilling kills jobs that exceed their estimates, so as not to violate subsequent commitments. This policy supposedly provides motivation for users to supply accurate estimates, because jobs would have a better chance to backfill if their estimates are tight, but would be killed if they are too short. Nevertheless, estimates are inaccurate despite this incentive, as depicted in Fig. 1.4 (page 13), which shows a uniform-like accuracy histogram when only considering jobs that have terminated successfully, meaning any level of accuracy is almost equally likely to happen. A possible reason is that users find the motivation to overestimate — so their jobs will not be killed — much stronger than the motivation to provide accurate estimates and help the scheduler to perform better packing. But a recent study indicates that users are actually quite confident of their estimates, and most probably would not be able to provide much better information [93]. As mentioned in the previous chapter, estimates also embody a characteristic that is particularly harmful for backfilling: they are inherently modal, as users tend to choose "round" estimates (e.g. one hour) resulting in 90% of the jobs using the same 20 values; worse, the most popular estimate is typically the maximal allowed. This significantly limits the scheduler's ability to exploit existing holes in the schedule because all jobs appear the same, and often too long. The combination of inaccuracy and modality deteriorates performance (Fig. 1.2, page 9; compare "real" to "perfect") and motivates searching for an alternative.

The alternative The search for better estimates has focused on using historical data in an attempt to predict the future, based on the fact users of parallel machines tend to repeatedly do the same work (Fig. 1.6, page 14). Suggested prediction schemes include using the top of a 95% confidence interval of job runtimes [62], a statistical model based on the (usually) log-uniform distribution of runtimes [31], using the mean plus 1.5 standard deviations [108], and more sophisticated techniques [62, 136, 83, 86, 96]. Despite all this work, production backfill schedulers in actual use still employ user estimates rather than history-based system-generated predictions, due to three

difficulties: (1) a technicality, (2) a usability issue, and (3) a misconception, to be further discussed next. *This chapter is about refuting or dealing with these difficulties*.

Technicality The core problem is that it's simply impossible to naively replace estimates with system predictions, as these might turn out too short leading to premature killing of jobs according to the backfilling rules. Suggested solutions have included ignoring the problem, using preemption, employing test runs, or replacing backfilling by shortest job first (SJF) [62, 174, 115, 19, 15, 90].[1] None of these retain the appeal of plain EASY. Mu'alem and Feitelson checked the extent of the underprediction phenomenon, showed it to be significant, and concluded that "it seems using system-generated predictions for backfilling is not a feasible approach" [108] (denoted the "unfeasibility claim" in Chapter 1). However, as we will show, solving this problem is actually quite simple: user estimates must serve as kill times (part of the user contract), while system predictions can be used for everything else.

Usability Previous prediction techniques have assumed that an important component is to identify the most similar jobs in the history, and base the predictions on them. To this end they employed complex algorithm including various statistical methods [31, 136, 97], genetic algorithms [136], instance based learning [83], and rough set theory [86]. In addition to their unwarranted computational overhead and complexity, most algorithms require a training period which can be significant, e.g. Smith et al. trained their algorithm using an entire trace before evaluating it (on the very same trace) [138]. In contrast, in this chapter we show that trivial algorithms (e.g. using the average runtime of two preceding jobs by the same user) can significantly improve the performance as well as the accuracy of the predictions themselves. We preferred using a simple predictor so as to focus on how predictions are integrated into backfilling schedulers, and not on the prediction algorithm itself. However, our evaluations indicate that this was a fortuitous choice, and that recency is actually more important than similarity when using historical data.

Misconception As noted above, studies regarding the impact of inaccuracy have found that it doesn't effect or even improves performance [146, 47, 169, 170, 34, 64], which has led to the suggestion that estimates should be doubled [174, 108] or randomized [115], to make them even less accurate. Doubling indeed exhibits remarkable improvements, which supposedly negates the motivation to improve the quality of estimates, deeming them as "unimportant". We show this to be false in three respects. First, we have already noted that while doubling original estimates helps, doubling of accurate estimates is even better (Fig. 1.2, page 9; compare "realX2" to "perfectX2"). In this chapter we show that doubling of good predictions is similar, namely, that the more accurate the original predictions are, the more the doubling is effective. Second, Chapter 3 has shown that the reason doubling helps is due to the "heel and toe" dynamics (Sec. 3.4), which trades off FCFS-fairness for performance by implicitly nudging the systems towards a more SJF-like schedule. (Incidentally, most studies dealing with predictions indicate that increased accuracy improves performance when shorter jobs are favored [62, 138, 174, 115, 15].) This chapter shows this tradeoff can be largely avoided by explicitly using a *shortest job backfilled first* (SJBF) backfilling order. By still preserving FCFS *reservation-order*, we maintain EASY's appeal, enjoying a fair scheduler that nevertheless backfills effectively. The third fallacy in the "inaccuracy helps" myth is that it implies predictions are only important for backfilling, even though they are used in other contexts as well (e.g. advance reservations for grid allocation and co-allocation, shown to considerably benefit from better accuracy [83, 137, 96]; or the scheduling of *moldable* jobs that

[1] Smith et al. didn't specify how they utilized system predictions for backfilling [138].

may run on any number of nodes [31, 138, 22], mandating the system to decide whether waiting for more nodes to become available is preferable over running immediately on what is currently available).

Naming Notation In our terminology, the term “estimate” always refers to the runtime approximation that was provided by the user upon job submittal. The term “prediction”, however, is overloaded. In its general meaning, prediction refers to the value that the system eventually uses for backfilling. When no system-generated predictions are employed, estimates and predictions are one and the same (e.g. for vanilla EASY that directly utilizes user estimates for backfilling). But when system-generated predictions come into play, this is no longer the case, and predictions and estimates may obviously differ. The second (more frequent) use of “prediction” is shorthand for “system-generated predictions”. The ambiguity is always resolved by the context in which the term is used.

Measuring Accuracy The measure of *accuracy* is the ratio of the real runtime to the prediction. If the prediction is larger than the runtime, this reflects the fraction of predicted time that was actually used. But as noted, predictions can also be too short. Consequently, to avoid under- and over-prediction canceling themselves out (when averaged), we define

$$accuracy = \begin{cases} 1 & \text{if } P = T_r \\ T_r/P & \text{if } P > T_r \\ P/T_r & \text{if } P < T_r \end{cases}$$

where P is the prediction; the closer the accuracy is to 1 the more accurate the prediction. This is averaged across jobs, and also along the lifetime of a single job, if the system updates its prediction. In that case a weighted average is used, where weights reflect the relative time that each prediction was in effect. More formally, given a job J, its weighted accuracy is $\sum_{i=1}^{N} A_i \cdot \left(\frac{T_i - T_{i-1}}{T_N - T_0}\right)$ where T_0 and T_N are J’s submission and termination time, respectively, and A_i is the accuracy of the prediction of J that was in effect from time T_{i-1} to time T_i.

Roadmap The rest of the chapter is structured as follows. Sec. 4.2 explains how prediction-based backfilling is done and demonstrates the improvements in terms of average performance and accuracy. Sec. 4.3 deals with “predictability”, namely, how do reservations relate to actual start times. Sec. 4.4 shows the generality of our techniques by applying them to schedulers other than EASY. We then discuss the connection between accuracy, performance, and predictability (Sec. 4.5). And finally, we investigate the optimal parameter settings for our prediction algorithms, and contrast the common approach of favoring similar jobs when generating predictions, with our approach of favoring recent ones (Sec. 4.6).

4.2 Incorporating Predictions into Backfilling Schedulers

The simplest way to incorporate system-generated predictions into a backfilling scheduler is to use them in place of user-provided estimates. The problem of this approach is that aside from serving as a runtime *approximation*, estimates also serve as the runtime *upper-bound* (kill-time). But predictions might happen to be shorter than actual runtimes, and users will not tolerate their jobs being killed just because the system speculated they were shorter than the user estimate. So it is not advisable to just replace estimates by predictions. Previous studies have dealt with

trace	wait [minutes]			b. slowdown			accuracy [%]		
	$EASY$	$EASY_{PRED}$		$EASY$	$EASY_{PRED}$		$EASY$	$EASY_{PRED}$	
SDSC	363	757	+109%	99	233	+136%	32	55	+70%
CTC	21	29	+38%	4.6	7.0	+53%	39	56	+44%
KTH	114	968	+748%	90	746	+729%	47	49	+4%
BLUE	130	1324	+920%	35	439	+1141%	31	55	+78%
avg.			+454%			+515%			+49%

Table 4.1: *Average wait time (minutes), bounded slowdown, and accuracy for vanilla backfilling with user estimates (EASY), and when these are replaced by our simple system-generated predictions ($EASY_{PRED}$). Shaded columns give changes relative to EASY in percents. These are always positive, which is a good thing for accuracy (as now it is higher), but bad for the other metrics (bigger wait period and slowdown).*

this difficulty either by eliminating the need for backfilling (e.g. using pure SJF [62, 138]), by employing test runs [115, 15, 90], by assuming preemption is available (so jobs that exceed their prediction can be stopped and reinserted into the wait queue [62, 19]), or by considering only artificial estimates generated as multiples of actual runtimes (effectively assuming underprediction never occurs) [174, 115, 15, 141, 142]. As mentioned earlier, Mu'alem and Feitelson noted this problem, and investigated whether underprediction does in fact occur when using a conservative predictor (average of previous jobs with the same user / size / executable, plus $1\frac{1}{2}$ times their standard deviation) [108]. They found that ∽20% of the jobs suffered from underprediction and would have been killed prematurely by a backfill scheduler. They therefore suggested that system-generated predictions for backfilling is not a feasible approach.

4.2.1 Separating the Dual Roles of Estimates

The key idea of our solution is recognizing that the underprediction problem emanates from the dual role an estimate plays: both as a prediction and as a kill-time. We argue that these should be separated. It is legitimate to kill a job *once its user estimate is reached*, but not any sooner; therefore user estimates should only retain their role as kill-times. All the other considerations of a backfilling scheduler should be based upon predictions, which can potentially be more accurate. There is no technical problem preventing us from running any backfill scheduler using predictions instead of estimates. The only change is that a running job is *not* killed when its prediction is reached; rather, it is allowed to continue, and is only killed when it reaches its estimate. This entirely eliminates the problem of premature killings.

The system-generated prediction algorithm we use is very simple. The prediction of a new job J is set to be the average runtime of the two most recent jobs that were submitted by the same user prior to J and that have already terminated. If no such two jobs exist we fall back on the associated user estimate (other ways to select the history jobs are considered in Sec. 4.6). If a prediction turns out higher than the job's estimate it is discarded, and the estimate is used, because the job would be killed anyway when it reaches its estimate. Implementing this predictor is truly trivial and requires about a dozen lines of code: saving the runtime of the two most recent jobs in a per-user data structure, updating it when more recently submitted jobs terminate, and averaging the two runtimes when a new job arrives. Nevertheless, as shown below, this simple predictor is enough to significantly improve the accuracy of the data used by the scheduler, which is sufficient for our needs in this chapter.

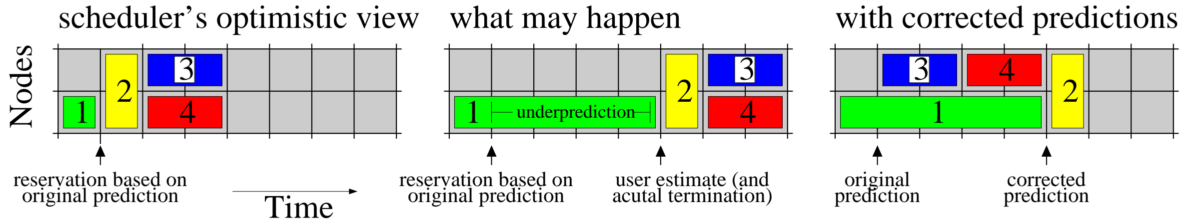


Figure 4.1: *Underpredicting the runtime of job 1 causes the scheduler to make an early reservation for job 2 (left). This misconception prevents jobs 3 and 4 from being backfilled (middle). Correcting the prediction once proved wrong enables the scheduler to reschedule the reservation and re-enables backfilling (right).*

Tab. 4.1 shows the results of our experiment of running a system using original EASY vs. a system in which estimates are replaced with our automatically generated predictions. The results indicate a colossal failure. Both performance metrics (average wait and slowdown) consistently show that using predictions results in severe performance degradation of up to an order of magnitude (KTH's wait time). This happens despite the improved accuracy of the predictions. The first suspect of being responsible for the dismal results was of course our ridiculously simplistic prediction algorithm. However, as noted, even these simple predictions are usually far superior to the estimates supplied by users, and may almost double the average accuracy. Discovering the underlying reason for the performance loss required a thorough investigation. Our in-depth analysis revealed that the true guilty party is *underestimation*, that is, cases in which a generated prediction is smaller than the job's actual runtime. This problem is addressed next.

4.2.2 Prediction Correction

By the rules of backfilling, a reservation computed based on user estimates will never be smaller than the start time of the associated job, as estimates are runtime upper bounds.[2] This is no longer true for predictions, as they are occasionally too short. At the extreme, predictions might erroneously indicate that certain jobs should have terminated by now and thus their processors should be already available. Assuming there aren't enough processors for the first queued job J, this discrepancy might lead to a situation where J's reservation is made for the present time, because the scheduler erroneously thinks the required processors should already be available.

Note that the backfill window is between the current time (lower bound) and the reservation (upper). When these are made equal, backfill activity effectively stops[3] and the scheduler largely reverts to plain FCFS, eliminating the potential benefits of backfilling (Fig. 4.1, left/middle). This explains why our naive approach from above dramatically worsened performance, despite the improvement in average accuracy.

The solution is to modify the scheduler to increase expired predictions proven to be too short. For example, if a job's prediction indicated it would run for 10 minutes, and this time has already passed but the job is still running, we must generate a new prediction. The simplest approach is to acknowledge that the user was smarter than us and set the new prediction to be the user's estimate (other approaches are explored later on). Once the prediction is updated, this affects reservations

[2]Apparently, this is not always the case in practice, as will shortly be described.

[3]The only remaining backfill activity is on the expense of the "extra" processors, which are the "leftover" after satisfying the reservation for the first queued job, see Fig. 3.14 (page 49).

trace	*underestimated jobs*	
	number	*%*
SDSC	4,138	7.7%
CTC	7,174	9.3%
KTH	478	1.7%
BLUE	22,216	9.9%

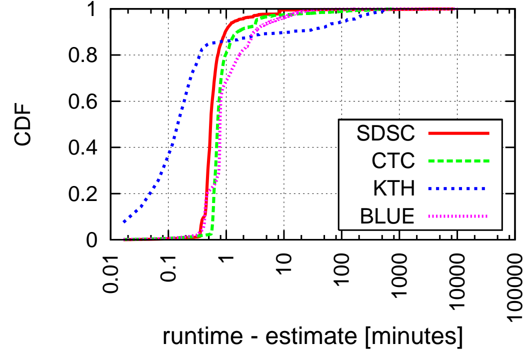


Figure 4.2: *Left: up to 10% of the jobs have runtimes bigger than user estimates. Right: CDF of differences between runtimes and estimates, of underestimated jobs. Most estimate violations are less than one minute.*

for queued jobs and re-enables backfilling (Fig. 4.1, right). While this may undesirably delay the reservations made for queued jobs, such delays are still bounded by the original runtime estimates of the running (underpredicted) jobs.

On rare occasions prediction correction is necessary even beyond the estimate, as in real systems jobs sometimes exceed their estimates (Fig. 4.2, left). In most cases the overshoot is very short (not more than a minute) and probably reflects the time needed to kill the job. But in some cases it is much longer, for unknown reasons. Regardless of the exact reason, the prediction should be extended to keep the scheduler up to date (independent of the fact the job should be killed, and maybe *is* being killed). As most of these jobs only exceed their estimate by a short time, we enlarge post-estimate predictions in a gradual manner: The first adjustment adds only one minute to the old prediction. This will cover the majority of the underestimated jobs (Fig. 4.2, right). If this is not enough, the ith prediction correction adds $15 \times 2^{i-2}$ minutes (15min, 30min, 1h, 2h, etc.).

The results of adding prediction correction are shown in Tab. 4.2. This compares the original EASY with a version that uses user estimates as predictions and adds prediction correction (EASY$_{PCOR}$), and a version that combines prediction correction with system-generated predictions (EASY^{+}). Note that while EASY$_{PCOR}$ employs user estimates as predictions, correction is still needed to handle the underestimated jobs discussed earlier. Prediction-correction by itself has only a marginal effect, because only a small fraction of the jobs are grossly underestimated. The real value of prediction correction is revealed in EASY^{+}, where system-generated predictions are added: results show a significant and consistent improvement of up to 28% (KTH's slowdown in Tab. 4.2). This is an important result that shouldn't be taken lightly. The fact that historical information can be successfully used to generate runtime predictions is known for more than a decade [48]. Our results in Tab. 4.2 demonstrate for the first time that this may be put to productive use within backfilling schedulers, without violating the contract with users. Moreover, the overhead is low, with predictions corrected only 0.56–0.63 times on average per job.

Note that obtaining the reported improvement is almost free. All one has to do is create predictions as the average runtime of the user's two most recent jobs and set an alarm event to correct those predictions that prove too short. Importantly, this does not change the way users view the scheduler, allowing the popularity of EASY to be retained. Finally, note that this scheme significantly improves the average accuracy, which can be up to doubled (BLUE) and is stabilized at 60–62% across all four traces when using EASY^{+}.

trace	wait [minutes]					b. slowdown					accuracy [%]					avg. corr. [$\pm\sigma$]	
	$EASY$	$EASY$ $PCOR$		$EASY^+$ all		$EASY$	$EASY$ $PCOR$		$EASY^+$		$EASY$	$EASY$ $PCOR$		$EASY^+$		$EASY$ $PCOR$	$EASY^+$
SDSC	363	360	-1%	326	-10%	99	93	-6%	86	-13%	32	32	+0%	60	+87%	$0.09_{\pm 0.33}$	$0.56_{\pm 0.64}$
CTC	21	21	+0%	16	-26%	4.6	4.5	-2%	3.3	-27%	39	39	+0%	62	+61%	$0.12_{\pm 0.41}$	$0.63_{\pm 0.69}$
KTH	114	115	+1%	96	-16%	90	90	+1%	65	-28%	47	47	+0%	60	+28%	$0.02_{\pm 0.24}$	$0.53_{\pm 0.57}$
BLUE	130	128	-1%	102	-21%	35	36	+1%	26	-25%	31	31	+0%	61	+100%	$0.13_{\pm 0.48}$	$0.60_{\pm 0.69}$
avg.			-0%		-18%			-2%		-23%			+0%		+69%	0.09	0.58

Table 4.2: *Average performance, accuracy, and overhead for scheduler variants. $EASY_{PCOR}$ adds prediction correction (needed even when user estimates serve as predictions, as these are occasionally smaller than runtimes). EASY$^+$ further adds system-generated predictions, replacing estimates. As before, shaded percents are changes relative to EASY; negative values are good for wait/slowdown; positive ones are good for accuracy. Right most metric shows the per-job average prediction-correction number ($\pm$ std. deviation).*

4.2.3 Shortest Job Backfilled First (SJBF)

A well known scheduling principle is that favoring shorter jobs significantly improves overall performance. Supercomputer batch schedulers are one of the few types of systems which enjoy a-priori knowledge regarding runtimes of scheduled tasks, whether through estimates or predictions. Therefore, SJF scheduling may actually be applied. Moreover, several studies have demonstrated that the benefit of accuracy dramatically increases if shorter jobs are favored [62, 138, 174, 115, 15, 131]. For example, Chiang et al. [15] show that when replacing user estimates with actual runtimes, while ordering the wait queue by descending $\sqrt{\frac{T_w+T_r}{T_r}} + \frac{T_w}{100}$, average and maximal wait times are halved and slowdowns are an order of magnitude lower.[4]

Contemporary schedulers such as Maui can be configured to favor (estimated) short jobs, but their default configuration is essentially the same as in EASY [37] (SJF is the default only in PBS). This may perhaps be attributed to a reluctance to change FCFS-semantics perceived as being the most fair. Such reluctance has probably hurt previously suggested non-FCFS schedulers, that impose the new ordering as a “package deal”, affecting both backfilling and reservation order (for example, with SJF, a reservation made for the first queued job helps the shortest job, rather than the one that has been delayed the most). In contrast, we suggest separating the two.

Our scheme introduces a controlled amount of “SJFness”, but preserves EASY’s FCFS nature. The idea is to keep reservation order FCFS (as in EASY) so that *no job will be backfilled if it delays the oldest job in the wait queue*. In contrast, backfilling is done in SJF order, that is, Shortest Job Backfilled First — SJBF. This is acceptable, as the first-fit essence of backfilling is a departure from FCFS anyway. We argue that in any case, explicit SJBF is more sensible than “tricking” EASY into SJFness by doubling [174, 108] or randomizing [115] estimates (see Sec. 4.5).

Results of applying SJBF are shown in Tab. 4.3. In its simplest version this reordering is used with conventional EASY (i.e. using user estimates and no prediction correction). Even this leads to typical improvements of 10–20%, and up to 42% (BLUE’s bounded slowdown).

Much more interesting is EASY^{++} which adds SJBF to EASY$^+$ (namely combines system-generated predictions, prediction correction, and SJBF). This usually results in double to triple

[4] Recall that T_w and T_r are wait- and run-times. Short jobs are favored since the numerator of the first term rapidly becomes bigger than its denominator. The second term is added in an effort to avoid starvation. We remark that this priority is very similar to the LXF&W priority used in Chapter 3, which was proposed by the same researchers.

trace	wait [minutes]						
	EASY	EASY SJBF		EASY++		PERFECT++	
SDSC	363	361	-0%	327	-10%	278	-23%
CTC	21	19	-10%	14	-33%	19	-10%
KTH	114	102	-11%	95	-17%	91	-20%
BLUE	130	102	-21%	87	-33%	87	-33%
avg.			-10%		-23%		-22%

trace	b. slowdown						
	EASY	EASY SJBF		EASY++		PERFECT++	
SDSC	99	87	-12%	70	-29%	58	-42%
CTC	4.6	3.9	-14%	2.9	-37%	2.8	-39%
KTH	90	73	-19%	57	-36%	50	-44%
BLUE	35	21	-42%	19	-47%	13	-64%
avg.			-22%		-37%		-47%

trace	accuracy [%]						
	EASY	EASY SJBF		EASY++		PERFECT++	
SDSC	32	32	+0%	60	+87%	100	+211%
CTC	39	39	+0%	62	+61%	100	+158%
KTH	47	47	+0%	61	+28%	100	+111%
BLUE	31	31	+0%	62	+102%	100	+225%
avg.			+0%		+70%		+176%

Table 4.3: *Average wait, bounded slowdown, and accuracy of EASY compared with three improved variants. $EASY_{SJBF}$ just adds SJF backfilling (based on original user estimates). $EASY^{++}$ employs all our optimizations: system-generated predictions, prediction correction, and SJBF. $PERFECT^{++}$ is the optimum, using SJBF with perfect predictions. Shaded columns show improvement relative to traditional EASY.*

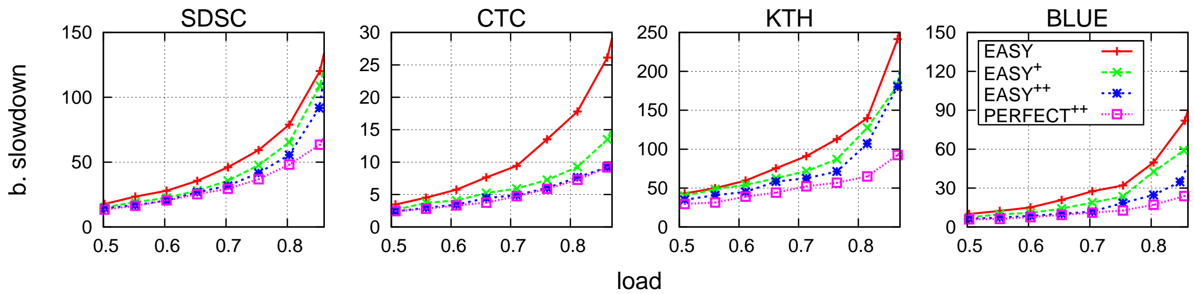


Figure 4.3: *Relative performance of $EASY^{+}$ / $EASY^{++}$ typically improves with medium or higher loads.*

the performance improvement in comparison to $EASY_{SJBF}$ and $EASY^{+}$. Performance gains are especially pronounced for bounded slowdown (nearly halved in BLUE). There is also a 33% peak improvement in average wait (CTC and BLUE). This is quite impressive for a scheduler with basic FCFS semantics that differs from EASY by only a few dozens lines of code. Even more impressive is the *consistency* of the results, which all point to the same conclusion, as opposed to other experimental evaluations in which results depended on the trace or even the metric being used [138, 39]. The accuracy of $EASY^{++}$ is similar to that of $EASY^{+}$ at 60–62%.

Finally, we have also checked the impact of having perfect predictions when SJBF is employed (here there is no meaning to prediction correction as predictions are always correct). It turns out $PERFECT^{++}$ is marginally to significantly better than $EASY^{++}$ with the difference being most pronounced in SDSC, the site with the highest load (Tab. 2.1; further discussed below). Interestingly, $EASY^{++}$ outperforms $PERFECT^{++}$ in CTC's average wait. This is due to subtle backfill issues and a fundamental difference between CTC and the other logs, as analyzed by Feitelson [39].

4.2.4 Varying the Load

All results in this chapter evaluate our suggested optimizations using the workloads "as is". Here, through trace manipulation, we complement our measurements by investigating the effect of load. Load is artificially varied by multiplying all arrival times by a constant (see Chapter 2). Results show that $PERFECT^{++}$ is better than $EASY^{++}$, which is better than $EASY^{+}$, which is better than EASY (Fig. 4.3). Higher loads usually intensify the trends pointed out earlier, but the precise effect of the optimizations is workload dependent. $EASY^{++}$ benefits are relatively small in SDSC, especially under high loads; for KTH the biggest improvement occurs for intermediate loads of

algorithm	*optimization*		
	prediction correction	*replace estimate with prediction*	*SJBF*
EASY			
EASY_{PRED}		√	
EASY_{PCOR}	√		
EASY_{SJBF}			√
EASY^{+}	√	√	
EASY^{++}	√	√	√
PERFECT^{++}	N/A	(with runtime)	√

Table 4.4: *Summary of the algorithms used in this section, and the optimizations they employ.*

around 70–80%; for CTC, the improvement over EASY grows with load, and is most significant towards 90%. Examining PERFECT^{++}, we see that in all cases accuracy becomes crucial as load conditions increase, generating a strong incentive for developing better prediction schemes.

4.2.5 Optimizations Summary

To summarize, three optimizations were suggested: (1) prediction correction where predictions are updated when proven wrong, (2) simple system-generated predictions based on recent history of users, and (3) SJBF in which backfilling order is shortest job first. All optimizations maintain basic FCFS semantics. They are all orthogonal in the sense that they may be applied separately. However, using system generated predictions without prediction correction leads to substantially decreased performance. The combination of all three consistently yields the best improvement of up to doubling performance in comparison to the default configuration of EASY. The algorithms covered and the optimizations they employ are summarized in Tab. 4.4 for convenience. The rest of the chapter will focus on EASY^{+} and EASY^{++}.

4.3 Predictability

Previous sections have shown that, on average, replacing user estimates with system-generated predictions is beneficial in terms of both performance and accuracy. However, when abandoning estimates in favor of predictions, we might lose *predictability*: The original backfilling rules state that a job J_b can be backfilled if its estimated termination time does not violate the reservation time R_1 of the first queued job J_1. Since J_b is killed when reaching its estimate, it is guaranteed that J_1 will indeed be started no later than R_1. However, this is no longer the case when replacing estimates with predictions, as R_1 is computed based on predictions, but jobs are not killed when their predicted termination time is reached; rather, they are simply assigned a bigger prediction. For example, if J_b is predicted to run for 10 minutes and R_1 happens to be 10 minutes away, then J_b will be backfilled, even if it was estimated to run for (say) three hours. Now, if our prediction turned out to be too short and J_b uses up its entire allowed three hours, J_1 might be delayed by nearly 3 hours beyond its reservation.

Predictability is important for two main reasons. One is the support of *moldable* jobs [31, 138, 22], that may run on any partition size (according to [23], ∽98% of the jobs are moldable). Such jobs trust the scheduler to decide whether waiting for more nodes to become available is preferable

trace	rate [% of jobs]					avg. diff. [minutes]					median diff. [minutes]					stddev diff. [minutes]				
	$EASY$	$EASY^{+}$		$EASY^{++}$		$EASY$	$EASY^{+}$		$EASY^{++}$		$EASY$	$EASY^{+}$		$EASY^{++}$		$EASY$	$EASY^{+}$		$EASY^{++}$	
SDSC	17	14	-18%	15	-15%	171	93	-46%	91	-47%	64	20	-69%	19	-70%	471	174	-63%	168	-64%
CTC	6.8	5.4	-19%	5.7	-16%	51	29	-43%	27	-46%	8.3	2.2	-73%	1.9	-78%	92	74	-20%	69	-25%
KTH	15	14	-8%	14	-8%	38	35	-7%	35	-7%	6.3	3.2	-49%	3.2	-49%	84	90	+7%	88	+5%
BLUE	9.6	7.5	-22%	7.8	-18%	68	45	-33%	45	-34%	16	3.3	-79%	3.4	-79%	212	191	-10%	184	-13%
avg.			-17%		-14%			-32%		-34%			-68%		-69%			-22%		-24%

Table 4.5: *Effect of predictions on the absolute difference between reservations and actual start times. Rate is the percentage of jobs that wait and get a reservation. Both rate and statistics of the distribution of differences are reduced with predictions, indicating improved performance and superior predictability, respectively.*

over running immediately on what's available now. Predictability is crucial for such jobs. For example, a situation in which we decide to wait for (say) 30 minutes because it is predicted a hundred additional nodes will be available by then, only to find that the prediction was wrong, is highly undesirable. The second reason predictability is important is that it is needed to support advance reservations. These are used to determine which of the sites composing a grid is able to run a job at the earliest time [96], or to coordinate co-allocation in a grid environment [83, 137], i.e. to cause cooperating applications to run at the same time on distinct machines. Note that in this case underprediction is as bad as overprediction, e.g. for a grid broker that must select where to dispatch a job. Knowing that resources would become available earlier could shift the balance.

The question is therefore which alternative (using estimates or predictions) yields more credible reservation times. To answer it, we have characterized the distribution of the absolute difference between a job's reservation and its actual start time. This is only computed for jobs that actually wait, become first, and get a reservation; jobs that are backfilled or started immediately don't have reservations, and are therefore excluded. A scheduler aspires to minimize both the number of jobs that need reservations and the differences between their reservations and start times. Note that with prediction correction a job may have multiple reservations during its life; we use the first for the predictability measurements.

The predictor we use (in this section only) is slightly different from the one used in Sec. 4.2: instead of using the last two jobs to make a prediction, we only use them if their estimate is identical to that of the newly submitted job; otherwise, we fall back on the user estimate. The reason is that, in some respects, this is the optimal predictor in this case; a full discussion of the tradeoffs along with results for the predictor used so far are given in Sec. 4.5. Results are shown in Tab. 4.5. Evidently, the rate of jobs that need a reservation is consistently reduced by 8–22% when predictions are used, indicating more jobs enjoy backfilling and reduced wait times. The rest of the table characterizes the associated distribution of absolute differences between reservations and start times. Both EASY^{+} and EASY^{++} obtain big reduction in the average differences: e.g. on SDSC, from almost 3 hours (171 minutes) to about an hour and a half (91 minutes). Reductions in median differences are even more pronounced: they are at least halved across all traces, with a 79% top improvement obtained by EASY^{++} on BLUE. The variance of differences is typically also reduced, sometimes significantly, with an exception of a 5–7% increase for KTH. The bottom line is therefore that using runtime predictions consistently and significantly improves predictability of jobs' starting time.

Improving the quality of reservations on average is desirable e.g. in grid context where it is

trace	rate [% of jobs]					avg. delay [minutes]					median delay [minutes]					stddev delay [minutes]				
	$EASY$	$EASY^+$		$EASY^{++}$		$EASY$	$EASY^+$		$EASY^{++}$		$EASY$	$EASY^+$		$EASY^{++}$		$EASY$	$EASY^+$		$EASY^{++}$	
SDSC	1.5	3.8	+149%	3.8	+150%	513	92	-82%	86	-83%	0.9	8.6	+896%	8.3	+859%	1442	223	-85%	206	-86%
CTC	0.7	1.3	+81%	1.3	+83%	72	37	-49%	34	-53%	1.9	3.0	+62%	2.6	+43%	119	102	-14%	94	-21%
KTH	0.1	1.8	+1518%	1.8	+1493%	58	52	-11%	44	-23%	0.7	11	+1541%	9.9	+1428%	108	107	-1%	87	-20%
BLUE	0.9	1.8	+97%	1.8	+102%	48	35	-28%	31	-35%	0.8	2.2	+174%	2.1	+165%	318	154	-52%	136	-57%
avg.			+461%		+457%			-42%		-48%			+668%		+624%			-38%		-46%

Table 4.6: *Effect of predictions on the delays beyond a job's reservation. With predictions, the rate and median delay are increased, but the average and standard deviation of delays are reduced.*

important to know that a job will start on time. (If sites A/B declare they can run a job in 5/10 hours from the current time, respectively, then obviously site A will be chosen; however, if it turns out that site B could have executed the job in only 10 minutes from the current time, and just didn't know about it because of low quality predictions, then obviously submitting the job to A was the wrong way to go.) However, it is conceivable some systems would care more about jobs being delayed beyond their reservation, than started earlier. Tab. 4.6 shows the rate of delayed jobs and the distribution of actual delays. Even with plain EASY 0.1–1.5% of the jobs are delayed, because (as reported earlier) jobs sometimes outlive their user estimates. Unfortunately, when predictions come into play, the delays become much more frequent and involve 1.3–3.8% of the jobs. On the other hand, both the average delay and its standard deviation are dramatically reduced, e.g. SDSC's average drops from about 8.5 hours (513 minutes) to less than 1.5 (86 minutes) and its standard deviation drops at a similar rate. Medians values, however, increase by up to an order of magnitude (KTH/SDSC), though in absolute terms they are all less than ten minutes. This indicates that EASY's delay-distribution is highly skewed and that our techniques curb the tail, at the expense of making short delays more frequent.

Nevertheless, there are two solutions for systems that do not tolerate delays. One is to employ double booking: leave the internals of the algorithms based on predictions, while reporting to interested outside parties about reservations which would have been made based on user estimates (never violated if jobs are killed on time). This solution enjoys EASY^{++}'s performance but suffers from EASY's (in)accuracy. The other solution is to backfill jobs in prediction order, but only if their user-estimated termination falls before the reservation. This ensures backfilled jobs do not interfere with reservations, at the price of reducing the backfilling rate. Indeed, this algorithm enjoys all the benefits of the "+" variants in terms of internal accuracy, while being similar or better than EASY with respect to unwarranted delays. As for performance, it is 1–10% better than that of plain EASY$_{SJBF}$ (Tab. 4.3).

4.4 Relationship With Other Algorithms

Our measurements so far have compared various scheduling schemes, culminating with EASY^{++}, against vanilla EASY. However, other variants of backfilling schedulers have been proposed since the original EASY scheduler was introduced. In this respect, it is desirable to explore two aspects: comparing EASY^{++} against some other generic proposals, along with investigating the effect of directly applying our optimization techniques to the other schedulers themselves.

We have chosen to compare EASY^{++} against the two generic scheduling alternatives that were previously mentioned in this chapter: EASY with doubled user estimates (denoted $X2$), and SJF

trace	wait [minutes]							b. slowdown							accuracy [%]						
	$EASY$	$X2$		SJF		$EASY^{++}$		$EASY$	$X2$		SJF		$EASY^{++}$		$EASY$	$X2$		SJF		$EASY^{++}$	
SDSC	363	333	-8%	535	+47%	327	-10%	99	89	-10%	69	-30%	70	-29%	32	16	-49%	32	-0%	60	+87%
CTC	21	20	-8%	13	-38%	14	-33%	4.6	4.1	-10%	2.8	-40%	2.9	-37%	39	20	-49%	39	+0%	62	+61%
KTH	114	102	-11%	79	-31%	95	-17%	90	80	-11%	45	-50%	57	-36%	47	24	-50%	47	-0%	61	+28%
BLUE	130	115	-11%	81	-38%	87	-33%	35	30	-15%	25	-29%	19	-47%	31	16	-47%	31	+0%	62	+102%
avg.			-10%		-15%		-23%			-12%		-37%		-37%			-49%		+0%		+70%

Table 4.7: *Average wait and bounded slowdown achieved by EASY^{++} compared with two other schedulers proposed in the literature: doubling user estimates and using SJF scheduling.*

trace	doubling																			shortest job									
	wait [minutes]									b. slowdown										wait [minutes]					b. slowdown				
	$X2$	$X2^+$		$X2_{perf}$		$X2^{++}$		$X2^{++}_{perf}$		$X2$	$X2^+$		$X2_{perf}$		$X2^{++}$		$X2^{++}_{perf}$		SJF	SJF^+		SJF_{perf}		SJF	SJF^+		SJF_{perf}	
SDSC	333	357	+7%	293	-12%	333	-0%	270	-19%	89	94	+6%	77	-13%	67	-25%	58	-34%	535	308	-42%	270	-50%	69	34	-51%	19	-73%
CTC	20	16	-18%	18	-8%	15	-25%	16	-16%	4.1	3.6	-13%	3.2	-21%	3.0	-28%	2.5	-38%	13	12	-11%	12	-11%	2.8	2.4	-12%	1.8	-35%
KTH	102	98	-4%	95	-6%	93	-8%	84	-18%	80	66	-18%	70	-13%	53	-33%	50	-38%	79	87	+10%	67	-16%	45	44	-2%	24	-46%
BLUE	115	105	-9%	107	-8%	86	-26%	80	-31%	30	33	+8%	28	-7%	21	-32%	12	-59%	81	90	+11%	50	-39%	25	37	+49%	5.4	-78%
avg.			-6%		-8%		-15%		-21%			-4%		-14%		-30%		-42%			-8%		-29%			-4%		-58%

Table 4.8: *Average performance and (shaded) improvement when optimizing vanilla $X2$ and SJF.*

based on user estimates (as a representative of several different schemes that prioritize short jobs). The results are shown in Tab. 4.7. EASY^{++} outperforms $X2$ by a wide margin for all traces and both metrics. It is also rather close to SJF scheduling in all cases, and outperforms it in one case (SDSC's wait) where SJF fails for an unexplained reason. The advantage over SJF is, of course, the fact that EASY^{++} is fairer, being based on FCFS scheduling with no danger of starvation. Also, the gap can potentially be reduced if better predictions are generated.

As mentioned earlier, EASY^{++} attempts to be similar to prevalent schedulers' default setting (usually EASY [37]) in order to increase its chances to replace them as the default configuration. But the techniques presented in this chapter can be used to enhance *any* backfilling algorithm. Tab. 4.8 compares vanilla $X2$ and SJF to their corresponding "optimized" versions: In addition to doubling of estimates (recall that these serve as fallback predictions when there's not enough history), $X2^+$ replaces estimates with (doubled) predictions, and employs prediction correction. $X2^{++}$ adds SJBF to $X2^+$. Finally, SJF$^+$ is similar to EASY^{++}, but allocates the reservation to the shortest (predicted) job, rather than to the one that has waited the most.[5] The theoretical optima of $X2^+$, $X2^{++}$, and SJF$^+$, are $X2_{perf}$, $X2^{++}_{perf}$, and SJF$_{perf}$, respectively (use perfect estimates instead of system-generated predictions).

Tab. 4.8 shows that switching from $X2$ to $X2^+$ can better performance (up to -18% in CTC's wait and KTH's slowdown) or worsen it (up to +8% in BLUE's slowdown), though improvements are more frequent and on average, $X2^+$ is 4-6% better than $X2$. When further optimizing by adding SJBF ($X2^{++}$), performance is consistently better, with a common improvement of 25-33%. The result of upgrading SJF to SJF$^+$ is once again inconsistent among traces/metrics, but here too improvements are more frequent (4-8% on average). In all cases, using prefect predictions ($X2_{perf}$, $X2^{++}_{perf}$, and SJF$_{perf}$) leads to consistent improvements in performance, indicating prior inconsistency steamed from our simplistic predictor and motivating the search for a better one.

[5] SJF$^+$ and SJF^{++} are equivalent because both employ SJBF by definition.

trace	performance						accuracy			prediction			py_{abs}						py_{delay}					
	wait [min]			b. slowdown						rate [job %]			rate [job %]			minutes			rate [job %]			minutes		
	imm	all		imm	all		imm	all		imm	all		imm	all		imm	all		imm	all		imm	all	
SDSC	363	327	-10%	81	70	-13%	56	60	+8%	59	89	+52%	15	12	-21%	91	98	+8%	3.8	5.0	+30%	86	150	+74%
CTC	17	14	-16%	3.6	2.9	-20%	59	62	+6%	63	90	+44%	5.7	5.1	-11%	27	27	-1%	1.3	1.7	+26%	34	53	+56%
KTH	98	95	-3%	67	57	-15%	58	61	+5%	39	84	+115%	14	11	-22%	35	63	+78%	1.8	3.6	+106%	44	149	+236%
BLUE	100	87	-13%	19	19	-2%	59	62	+5%	70	90	+29%	7.8	5.2	-33%	45	65	+46%	1.8	2.0	+10%	31	136	+333%
avg.			-10%			-12%			+6%			+60%			-22%			+33%			+43%			+175%

Table 4.9: *Comparing the immediate and all versions of EASY^{++}: py_{abs} relates to metrics from Tab. 4.5 (absolute difference between start time and reservation); py_{delay} relates to metrics from Tab. 4.6 (delay beyond a reservation). The all version is ∽10% better in terms of average performance and 6% more accurate. Nevertheless, despite its improved accuracy, it seems to loose in predictability: its py_{abs} rate is 11-33% lower (good), but the actual difference might be 78% higher (KTH); worse, both rate and duration of delayed jobs are significantly increased (KTH's rate is doubled, BLUE's delay is more than quadrupled).*

4.5 Does Better Accuracy Imply Better Performance/Predictability?

This study is based on the notion that superior accuracy should result in improved performance (better packing) and predictability (better individual runtime predictions). However, we have also witnessed several occasions in which these metrics appear to conflict. This issue has been largely dealt with in the previous chapter. But (1) in order to close some loose ends from previous sections, and (2) to argue that, similarly to multiplying of user estimates, multiplied predictions shouldn't be used as a model for "worse" predictors,[6] and (3) for completeness, we also conduct a short discussion here.

The **first** and most obvious (though already addressed) example is Fig. 1.2 (page 9), where deliberately making estimates less accurate by doubling them consistently improves performance. A **second** example is related to the predictor switch done in Sec. 4.3. Throughout this chapter we've used what we call an all prediction window, where the last two terminated jobs by the same user were used for prediction, regardless of their attributes. In contrast, in Sec. 4.3 we've used an immediate window, in which we generate a prediction only if these two jobs have user runtime estimates that are equal to that of the newly submitted job (i.e. they are "similar"). The fact of the matter is that all (which is more accurate) is better for performance, whereas immediate appears as better for predictability (Tab. 4.9). Further, a **third** example is that the performance of immediate-EASY^{+} and $X2$ is very similar (Tab. 4.10). These schedulers are identical in every respect, except EASY^{+} uses runtime predictions, whereas $X2$ uses something that is even less accurate than user estimates (user estimates that are doubled). The fact the two yield similar performance might prompt a reader (who did not fully absorb the implications of Chapter 3) to raise the question of whether it is worthwhile to even bother with runtime prediction.

This section addresses the question implied from the three examples, namely, what makes accuracy, performance, and predictability seem contradictory? Beginning with why doubling estimates helps performance, we simply note that this question was already addressed to its full in Chapter 3. The bottom line was that due to the heel-and-toe dynamics, what $X2$ is really doing is trading off FCFS-fairness for performance. Indeed, when doubling real user estimates the "heel and toe" effect is greatly amplified (Tab. 4.11).

[6] A methodological mistake already done by Guim et al. that followed up on our work [64].

trace	wait [minutes]			b. slowdown		
	imm	$X2$		imm	$X2$	
SDSC	343	333	-3%	92	89	-3%
CTC	18	20	+7%	3.7	4.1	+10%
KTH	108	102	-6%	79	80	+2%
BLUE	121	115	-4%	31	30	-3%
avg.			-2%			+2%

Table 4.10: $X2$ and immediate $EASY^+$ *yield similar performance despite the fact they are identical except the latter improves predictions whereas the former worsens them.*

trace	stalled rate [%]			stall time [min]			avg. thieves #		
	$EASY$	$X2$		$EASY$	$X2$		$EASY$	$X2$	
SDSC	7.2	11	+49%	91	137	+50%	1.9	2.1	+9%
CTC	9.1	11	+9%	35	50	+42%	2.5	2.8	+10%
KTH	7.0	11	+61%	51	107	+111%	1.6	2.3	+45%
BLUE	9.2	12	+29%	54	118	+119%	2.3	3.1	+33%
avg.			+39%			+80%			+24%

Table 4.11: *"Heel and toe" effect is amplified due to* $X2$. *Rate is the percent of jobs that had their earliest start time pushed back due to the effect, out of waiting jobs that got a reservation. Stall-time is the average period between a job's earliest start-time (computed according to perfect estimates) and its actual start-time. "Thieves" indicate the per-job average number of times the earliest start-time is pushed back (3 times for* J_3 *in Fig. 3.5, page 41).*

Based on this analysis, it should be clear that comparing between $X2$ and immediate EASY$^+$ (Tab. 4.10) is actually comparing between two different types of unrelated and *orthogonal* optimizations: favoring shorter jobs vs. improving predictions. Thus, as was well established in Chapter 3 in relation to user estimates, doubling of prediction should be viewed as a property of a scheduler, *not* the prediction algorithm. Indeed, both Fig. 1.2 and Tab. 4.8 indicate that doubling of improved predictions (whether perfect or based on history) yields better performance than when doubling the lower quality user runtime estimates. We argue that predictors should strive to make the best predictions they can, and leave the choice of whether to exercise the performance/fairness tradeoff to the scheduler, where it belongs. In any case, an evaluation of the implications of prediction mistakes, where "mistakes" actually means multiplying predictions [64], would simply be dominated by the tradeoff and will not teach us anything new.

The remaining open issue is that all, which is more accurate, seems less predictable than immediate in Tab. 4.9. Nevertheless, all is actually more predictable. First, consider py_{abs}. While the absolute difference under immediate is reduced, the rate of jobs that suffer such a difference is significantly higher. To see which of the two metrics have more impact (rate or difference), we computed the average difference with respect to *all* the jobs in the log (product of Tab. 4.9's "rate" and "minutes" columns, divided by 100). This reveals that all is actually more predictable than immediate in 3 out of the 4 logs.[7]

As for py_{delay}, we note that this metric is actually *very* problematic and should not be used alone. For example, $X2$ obviously reduces the accuracy of estimates, but has much lower py_{delay} than using the estimates as is, because it computes reservations based on unrealistically too-long predictions; tripling the estimates would make the effect even more pronounced. Likewise, immediate produces less predictions than all and therefore falls back on user estimates more often (Tab. 4.9's prediction rate). This explains why immediate is less accurate. Additionally, as estimates are bigger than predictions (by definition), immediate's reservations are further away in the future. In other words, py_{delay} is an unreliable predictability metric as it only accounts for "one side to the coin": jobs that run *later* than their reservation.

[7]The number associated with immediate are 13,1.5, 4.6, and 3.5 minutes for SDSC, CTC, KTH, and BLUE, respectively; the numbers for all are 11, 1.4, 6.7, 3.4.

4.6 Tuning Parameters

The EASY^{++} algorithm has several selectable parameters that may affect performance. We have identified seven parameters (formally defined later on) that are mainly concerned with the definition of the *history window*: which previous jobs to use, and how to generate the prediction. Some of these parameters have only two optional values, while others have a wide spectrum of possibilities.

To evaluate the effect of different settings, we simulated *all* 8,640 possible parameter combinations[8], henceforth called *configurations*, using our four different workloads. This led to a total of nearly 35,000 simulations (8,640 times the 4 traces),[9] where each simulation yielded two performance metrics (average wait and slowdown). Thus, each configuration (that is, parameter combination) is evaluated by eight trace/metric *testcases* ($\{$SDSC,CTC,KTH,BLUE$\}\times\{$wait,slowdown$\}$).

The results of the simulations indicate that the "performance surface" is extremely noisy. There are many different and seemingly unrelated configurations that achieve high performance, but there is no single configuration that is best for all eight testcases. In order to provide effective guidance in choosing the parameters we therefore performed a joint analysis of all the data. Our goal is to find the best configuration, where "best" means robust good performance under all eight testcases. We anticipate that such a configuration will also perform well under other conditions, e.g. with new workloads, as will be explained below.

The analysis is done as follows. We start by ranking all 8,640 configurations (parameter combinations) in two steps. First, we evaluate the "degradation in performance" of each configuration c under each trace/metric testcase t. This is done relative to the best performing configuration b for that testcase, as follows: let P_b and P_c be the performance of b and c under t, respectively, then c's degradation under t is defined to be $100\frac{P_c}{P_b} - 100$. Thus, each configuration is now characterized by eight numbers, reflecting its relative performance degradation under the eight testcases. In the second step we average these eight values and the configurations are ranked accordingly: the best configuration, with the *lowest* average performance degradation, has rank 1; the worst configuration has rank 8,640. Even with this ranking, the top configurations are rather diverse (Tab. 4.12; parameters will be discussed shortly).

It is important to note that our methodology is finding a *compromise* that reflects all eight testcases. For example, the top ranking configuration is not top-ranked for any of the testcases individually. Instead, it suffers a degradations ranging from 4.1% to 21.8% relative to the best configurations for each testcase. But its average degradation is only 9.4%, which is lower than the average of any other configuration.

Recall we are searching for *robust* configurations. This robustness should manifest itself by being immune to trivial changes and small modification. The top ranking configuration does not qualify as such: it uses 11 jobs for its prediction window, but when this value is replaced with 12, the associated configuration is ranked 1,295 and suffers *double* the average performance degradation. It would be ludicrous to assume 11 is a magic number and to recommend using it based on this analysis. We therefore search for a *contiguous subspace* within the configuration space (namely, a set "nearby" configurations), such that *all* its population yields good results.

The distributions of the different parameter values are shown in Fig. 4.4, and we now discuss

[8]Product of the number of different values each parameter may have. Following the left-to-right parameters' order in Fig. 4.4, this is: $3 \times 2 \times 2 \times 2 \times 4 \times 3 \times 30 = 8,640$; see detailed explanation below.

[9]Some combinations were actually equivalent and were therefore only done once; an example is making predictions using the average, median, maximum, or minimum of history jobs when there is only one history job.

rank	*average performance degradation*	*configuration*						
		window size	*window type*	*fullness*	*metric*	*fallback*	*propagation*	*prediction correction*
1	9.41%	11	all	partial	avg	est	yes	estimate
2	10.60%	3	ext	full	avg	rel	yes	estimate
3	10.84%	16	all	full	med	rel	yes	estimate
4	11.16%	21	all	partial	med	rel	yes	estimate
5	11.25%	10	all	full	med	est	yes	estimate
6	11.31%	4	ext	partial	min	rel	yes	estimate
7/8	11.47%	9	all	partial	med	rel/est	no	estimate
9	11.56%	2	ext	full	min	est	yes	estimate
10/11	11.61%	22	all	partial	med	rel/est	no	estimate
...	...	...	...	...	...	...	...	...
8640	239.88%	26	all	full	min	est	no	gradual

Table 4.12: *Top and bottom ranked configurations.*

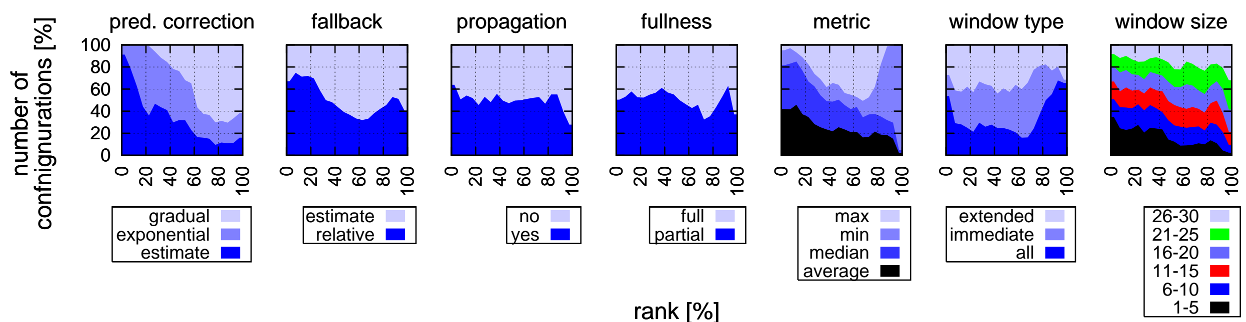


Figure 4.4: *Distributions of ranked configurations, as a function of each parameter. Rank values (X-axis) are converted to percents by dividing them with 8,640. Configurations are aggregated into 5%-sized bins. For example, with prediction correction (left subfigure), about 90% of the 5%-top-ranking configurations are associated with* estimate *and the remaining 10% are associated with* exponential*.*

each one in turn, starting with those that are easiest to characterize (left to right). The first parameter is how to perform **prediction correction** (when the predicted termination has arrived but the job continues to run). One option is to simply revert to the original user estimate. Other options are to grow the prediction gradually (by predefined increments as in Section 4.2), or in an exponential manner (by adding e.g. 20% each time). The results (Fig. 4.4, left) clearly indicate that it is best to jump directly to the full user estimate, and not to first try lower predictions, as this option dominates 90% of top-ranked configuration. This is probably so because using the full user estimate opens the largest window for backfilling. Using a gradual increase is especially bad, and dominates the bottom half of the ranked configurations.

When we cannot generate a prediction due to lack of historical information, we use the user estimate as a **prediction fallback**. The estimate can be used as is, or it can be relatively scaled according to the accuracy the user had displayed previously [136]. The results (Fig. 4.4, "fallback") show that relative provides a slight advantage, as it appears more often in high ranking configurations.

The next two parameters ("propagation" and "fullness") turned out not to have such decisive results, at least not when considered in isolation. **Propagation** refers to the action taken when new data becomes available. For example, if we make a prediction for a newly submitted job, and later

a previous job terminates, should we update the prediction based on this new information? The second is window **fullness**. The window is the set of history jobs that is used to generate predictions. The two options are to allow a partial window, meaning that a prediction is made based on whatever data is available, or to require a full window and use the user estimate as a fallback if not enough jobs are available. For both these parameters, the possible values are approximately evenly spread across the ranked configuration. The slight advantage of propagation seems not enough to justify its computational complexity. On the other hand, partial is significantly better when larger prediction windows are employed (not shown).

The last three parameters have intricate interactions that will eventually lead to the configuration subspace we seek. The first is the **prediction metric**. Given a set of history jobs, how should a prediction be generated? Four simple options are to use the average, median, minimum, or maximum of the runtimes of these jobs. Evidently, minimum tends to lead to a low-ranking configuration, and the maximum to a middle rank. The average and the median share 80% of the top-ranked configurations, leaving the question of which one should be used.

A harder question occurs with the **window type**. The three types are all, meaning that all recent jobs are eligible, immediate, meaning that recent jobs are used only if they are similar to the new job (same estimate), or extended, meaning similar jobs are used even if they are not the most recent (using the entire user history). The problem is that the all distribution has a U shape: it accounts for more than half the top-ranked configurations, but also for two-thirds of the lower-ranked ones.

Finally, a third difficult question is how to set the **window size** (the number of history jobs to consider). We simulated all sizes in the range 1–30; the graph (Fig. 4.4, right) shows them in bins of 5. Smaller windows are more common in high-ranking configurations, but there is no range-size that can be said to *dominate* high-ranking configurations.

To solve these problems we need to employ additional considerations, and to carefully study the interactions among the problematic parameters. We start with the window type parameter. There are actually big advantages to using the all window type. First, its evident top ranking peak. Second, it is easier and more efficient to implement, because we just need to keep a record of the runtimes of the last terminated (and most recently submitted) jobs by the user, and do not need to check for job similarity. The problem is that many configurations that employ an all window type are low ranking. The question is therefore whether we can avoid them (and how). Luckily, this can be done by a judicious choice of the other parameter values.

Specifically, there are 1298 configurations in the bottom-ranked 30% that employ an all window type. Of these, only 194 use the estimate directly as a prediction correction. As using estimate was shown above to be obviously beneficial, this helps eliminate 85% of the problematic configurations. Of the remaining configurations, 186 use the minimum prediction metric *and* employ a relatively large prediction window (≥ 7, with average of 18.8). It turns out the huge tail of minimum (Fig. 4.4) is mostly associated with large window sizes, and that increasing the window size consistently worsen the average degradation across *all* configurations. In fact, Fig. 1.19 (page 26) shows the connection between size and degradation is almost linear (both average and variance), with the exception that 2 is slightly better than 1.

The bottom line is that using an all window-type is actually safe in combination with estimate prediction correction and a small window size (< 7), eliminating more than 99% of all's tail configurations and clearly making it the best choice. Indeed, this subspace seems to meet our robustness demands, as is shown in Fig. 4.5 (left), because all its configurations are high ranking.

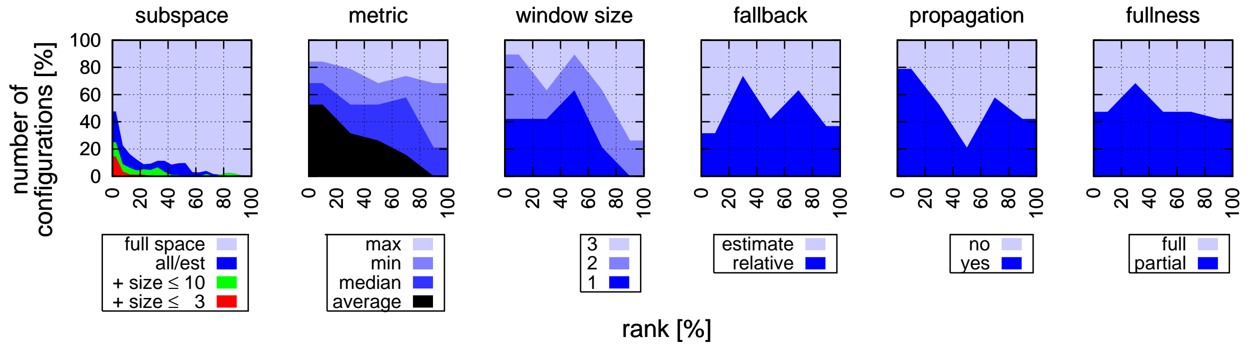


Figure 4.5: *Left: chosen subspace (type=*all*, correction=*estimate*) with decreasing window size. Others: parameters distribution within the smallest shown subspace (size $\leq$ 3).*

parameter	*description*	*suggestion*
window size	how many history jobs to consider	1-2
job selection	which jobs to include in the window	all
metric	how to generate predictions	average
correction	how to increase too-short prediction	user estimate

perf. degradation	*size=1*	*size=2*	*full space*
min	11.7%	12.0%	9.4%
max	15.3%	14.7%	239.9%
average	14.2%	13.4%	30.4%
std. deviation	1.2%	0.7%	10.1%

Table 4.13: *Suggested settings for EASY^{++} (left), and performance degradation statistics of size=1 / size=2 configurations within this chosen subspace, compared to the statistics of all 8,640 configurations (right).*

Accordingly, we choose to limit the window size of our chosen subspace to be $\leq$ 3. The rest of the sub-figures explore the remaining parameters within this subspace. Clearly, average is the preferable metric. Additionally, 1-2 sized windows are preferable over 3. However, it is hard to decide between the two because size=2 dominates the top (50%) but the worst-case of size=1 is better than that of size=2, and so we seem to have a tie. As there are also no clear winners within the other parameters, we conclude by summarizing our recommendations in Tab. 4.13, which match the prediction algorithms used in this chapter.

Note that our conclusions are in disagreement with previous work: Gibbons used all the history available [62], and Smith et al. experimented with a limited history just to reduce the size of the search space, implying a preference for the full history [136, page 129]. However, they did not show results. We too intuitively felt that when using historical information, it would be necessary to focus on *similar* jobs, i.e. those with the same partition size, executable, estimate, etc. This has motivated the definition of the extended window type. However, the results clearly show that *recency* is more important than similarity (Fig. 1.19, page 26) — it is better to use the last job by the same user than to search for the most similar job. This implies that the overheads for storing and searching through data about different classes of history jobs (as is done in e.g. [136, 138, 83, 86, 96]) can be avoided altogether. Arlitt et al. reached a similar conclusion in the context of the World Wide Web, contending "only the topmost stack element is seeing significant reuse" when predicting a destination of a work session based on the user's history stack[6]. Likewise, Isci et al. reached a similar conclusion when predicting the memory-references to operations ratio for the next 100-million operations phase: they concluded it's better to use the ratio of the last phase as a predictor, than to use an 8-sized prediction window or even a variable sized prediction window [71].

Finally, we remark that in addition to preferring to use all available history, Gibbons also used a different prediction metric: the 95th percentile of history jobs [62], which is close to the maximum metric, and was shown above to be inferior to the average.

4.7 Conclusions

For the conclusion of this chapter, we refer the reader to Section 7.3 (page 125).

Chapter 5

Modeling User Runtime Estimates

5.1 Introduction

Context This chapter was fully introduced in Sec. 1.2.2 (page 11) and Sec. 1.3.2 (page 21), which also conducted a detailed survey of related work. Briefly, backfill systems mandate users to provide runtime estimates for all the jobs they submit, which are then used by the scheduler for better packing. This was shown to improve utilization, throughput, and turnaround time. However, the strict policy of backfilling systems to (1) kill underestimated jobs, while (2) rewarding jobs with tight estimates (with increased chances for backfilling), has failed to deliver, as in spite of it, user estimates turned out to be highly inaccurate (Fig. 1.4, page 13). The previous chapters have thoroughly demonstrated that this inaccuracy has a decisive effect on performance, and that estimates are an important and influential parameter within the workload experienced by parallel machines. In fact, estimates are important enough to be considered alongside the three most important job attributes: arrival time, runtime, and size. The starting point of this chapter is noting that in contrast to the latter three attributes, which have been *extensively* studied and modeled [46, 30, 77, 17, 20, 170, 99, 105], estimates were largely neglected in this respect. The reason is most probably related to the misperception of inaccurate estimates as not affecting or improving performance, best articulated in a recent paper by England et al. [34] that claimed

ROBUSTNESS CLAIM

> *"Our results support those of a previous work and also indicate that backfilling is robust to inaccurate run time estimates in general. It seems that, with respect to backfilling, what the scheduler doesn't know won't hurt it."*

In Chapters 3 and 4 we have proved this claim, along with the aforementioned previous work, wrong. The goal of this chapter is therefore to close the modeling gap.

The Need For an Estimates Model The purpose of any good model, and hence an estimate-model, is to truthfully reflect reality. It is needed for three reasons. First, it is useful as part of a general workload model of parallel machines that allows for valid performance evaluations, e.g., to study different job scheduling schemes by means of simulation. (Simulations are particularly valuable when evaluating system designs; here, a model's main merit is its flexibility, allowing an evaluation of multiple configurations without having to actually build the respective systems.)

Second, it is often the case that estimates data is absent from existing log files of production systems [110], e.g., when those employed a non-backfill scheduler. Similarly to models, real logs

can also be used to drive simulations (the approach taken in our work). While not as flexible as models, real logs have the advantage of being truly representative of what really goes on in production systems, eliminating the concern that the chosen model might have neglected to reflect some important aspect.[1] An estimate model can artificially add the missing information to the log.

Indeed, an important product of our work in this chapter is a utility (available for download at [155]) that gets as input a log in "Standard Workload Format" [147] and appends to it estimates information. This utility can be used to complement both real production log-files and outputs of previously suggested models. Specifically, we advocate that this should replace the f-model and Φ-model (addressed below), which have been used for this purpose up till now.

The third reason for the need for a model, is that it allows for a deeper understanding of the nature of the workload, with which the associated systems must cope. This can often provide helpful insights as to "what went wrong" and as to how can a system design be improved to rectify the problem. To illustrate this, note that while Chapter 4 (that deals with improving the system rather than understanding it) appears before this chapter, chronologically, the order was reversed (see publication dates of [157] and [156]). Indeed, only after we modeled the estimates and understood their modal nature and its dire implication on backfilling, did we come up with the idea presented in Chapter 4, about how to improve estimates' quality in a manner usable for backfilling. The *real* motivation for Chapter 4 and its simplistic prediction approach was our understandings based on this chapter: that, in contrast to previous work [62, 83, 138, 86, 97], one in no way needs a sophisticated algorithm to overcome modality; rather, even a simple average of recent jobs will do. After this, it became clear that the focus should not be on the prediction algorithm, but rather, on how to exploit its output for the sake of backfilling. The bottom line is that only through *understanding* the workload were we able to understand how to improve the design.

We therefore note that, in contrast to the previous chapter, this one targets the first two above reasons. Namely, here we aim to understand, model, and reflect reality, *not* to make it better.

The Need For Estimates Altogether As noted above, the quality of estimates has a dramatic effect on the performance of systems, which makes them an important and model-worthy attribute. However, Chapter 4 have opened an intriguing possibility to rid users from the annoying need to provide estimates altogether. While our prediction algorithms above still make use of estimates for prediction-correction and as fallback when not enough history is available, it is technically possible to get by without this information. (We explore this idea in [152], but the details are beyond the scope of this dissertation). Nonetheless, we argue that systems with no user estimates *at all* (that is, with no runtime upper bound) are undesirable, as these will allow jobs to run indefinitely, potentially overwhelming the system (e.g. consider runaway buggy jobs). Thus, it appears such a policy is inadequate for supercomputers. At the very least we would expect users to choose some runtime upper-bound from a predefined set of values (possibly associated with different system queues). However, we will show below that this scenario is rather similar to reality anyway, as most users are already limiting themselves to very few canonical "round" estimates. It turns out there is actually no fundamental difference between allowing users to choose "any value", or from within a limited set. The bottom line is that regardless of any possible scheduling improvements or changes, it seems a parallel workload model will not be complete if realistic user estimates are not included.

[1] This concern is acute: suggested models were often found inadequate or lacking after they were put into use. One example is Lublin's model of rigid jobs [99], which Talby et al. found to lack self-similarity [151].

5.2 Existing Estimate Models and Their Shortcomings

As estimates were never "natively" incorporated within any previously-suggested general workload model, researchers were forced to artificially generate estimates by themselves. This was done using three models that specifically targeted estimates directly (runtimes, sizes, and arrival times came from a different source: either from the output of a general model, or from a real trace-file generated by a production system).

Complete Accuracy The simplest model is to assume that user estimates are completely accurate. For example, such a model was used by [84, 145, 163, 39, 133]. This approach has two advantages: it is extremely simple, and it avoids the murky issue of how to model user estimates correctly. However, as witnessed by the data in Fig. 1.14 (page 22) and Fig. 1.4 (page 13), it is far from the truth.

The f-Model The second model, which is a generalization of the first, is the f-model [47]. This model assumes a job's estimate is uniformly distributed in $[r, (f+1)\cdot r]$, where r is the job's runtime, and f is a non-negative "badness" factor. Thus, with $f = 0$, we get the above model, but increasingly positives fs supposedly model increasingly inaccurate estimates. This model proved to be the most popular and was extensively used to (1) study the impact of inaccuracy on performance [146, 174, 108, 15, 142, 122, 34, 64], and to (2) complement workloads that did not contain estimates data but were simulated under backfilling system [169, 56, 58].

By Chapter 3, the generalized f-model is actually worse than the previous model ($f = 0$), as positive fs suffer from all the shortcomings of a zero f, and more. The bottom line was that the f-model's artificial-inaccuracy simply trades off fairness for performance, which at best is reflective of a scheduling policy, but is anything but reflective or the impact of real estimates on performance (Tab. 3.17, page 52).

The Φ-Model The third model is the Φ-model [108], which attempts to reproduce the accuracy histogram in Fig. 1.4 (page 1.4). This model was described in detail in Sec. 1.2.2 (page 11). The flat histogram portion implies that $r/e = u$, for estimate e, and a uniform random variable $u \in [0, 1]$. Thus, e is modeled by r/u. The model also artificially created the hump of failed jobs at low values, and the 100% peak of underestimated job (the height of this peak was denoted Φ, and was set to be a parameter of the model [171]). Although far less popular than the f-model, the Φ-model was used for the same purposes, namely to complement workloads that were missing estimates data, and to evaluate the impact of increased inaccuracy (by gradually reducing Φ) [171, 170]. Similarly to the f-model, the Φ-model ignored the per-site limit on runtimes and estimates — E_{max}, which contradicts its assumption that r is independent of u: since estimates are bigger than runtimes, we have $r \le e \le E_{max}$, which means the bigger the r the more u $(= r/e)$ gets closer 1. In other words, r and u are proportional. And since u is defined to be the accuracy, longer jobs (bigger r) are always on the right of Fig. 1.4 (page 13), where accuracy is high, while short jobs tend to be on the left, at lower accuracies.

We note in passing that Cirne and Berman [20] suggested a model that takes the opposite direction in comparison to the Φ-model (see details in Sec. 1.2.2). They assumed e is given and that the model should generated r. This methodology suffers from the same problem as the original model, because accuracy is again independent of runtime.

Lack of Repetitiveness In addition to the per-model shortcomings listed above, there are two drawbacks from which all models collectively suffer: The first is lack of repetitiveness: The work

of users of parallel machines usually takes the form of temporal bursts of very similar jobs, characterized as "sessions" [173, 133]. In SDSC for example, the median value of the number of different estimates used by a user is only 3, which means most of the associated jobs look identical to the scheduler. In the next chapter (and in [160, 54]) we show that such repetitiveness can have decisive effect on performance.

Lack of Modality The second shortcoming of all models is a direct result of the first: estimates form a modal distribution composed of very few values, a fact that is not reflected in any existing model. Modality was shown above to be particularly harmful in the context of backfilling. While this was largely revealed during the development of the model to be described in this chapter, we already considered the issue in great detail (Section 3.8, page 48). Thus, we only conduct a short discussion here regarding the bottom line.

Jobs that use the maximal allowed estimate — E_{max} — cannot be backfilled (see proof in aforementioned section). Therefore, the fact E_{max} is always a very popular estimate (typically the most; see Fig. 1.14, page 22) has a detrimental effect on performance. At the extreme, if all jobs used E_{max}, backfilling activity would completely stop (except from on the "extra" nodes; see Fig. 3.14 page 49) and the schedule would largely revert to plain FCFS. The observation regarding E_{max} is true to some extent for all the other modes (= popular estimates), as in general, if the estimates distribution is dominated by only a few large modes, performance is negatively effected, because less variance among jobs means less opportunities for the scheduler to perform backfilling using existing holes. In addition, many scheduling policies provide some preference to shorter jobs. This may be an explicit preference, as is the case of the LXF&W scheduler (discussed in Sec. 3.10, page 55), or an implicit preference, as is the case of plain EASY, due to users' inaccurate estimates and the resulting heel and toe effect. Obviously, the more the estimates distribution is modal, the less the scheduler can distinguish between short and long jobs, and performance deteriorates accordingly.

The result of replacing real user estimates by values that were generated by the above three models is shown in Fig. 1.5 (page 13). It is evident all of models produce unrealistically good results in comparison to the original. While counter intuitive, our goal in this chapter is to produce estimates such that performance is *worsened*, not improved. Namely, our aim is to accurately reflect reality, not to paint a brighter (false) picture.

Arbitrary Binning When addressing modality, an immediate heuristic that comes to mind when trying to reconstruct this property within a model, is to "round" artificially generated estimates (e.g. by one of the models described above) to the nearest "canonical" value: values smaller than 1 hour are rounded to (say) the nearest multiple of 5 minutes, values smaller than 5 hours are rounded to the nearest hour, and so on. Experiments have shown that this heuristic also fails in capturing the badness of real user estimates, and performance results are actually rather similar to those obtained before this artificial modality was introduced.

Further, arbitrary "binning" fails to reproduce the various properties of the estimate distribution, as reported in the following sections. The fact of the matter is that modes have a different (worse) nature than produced by the arbitrary binning. For example, when examining the number of jobs associated with the most popular estimates, we learn that these decay in an exponential manner e.g. half of the jobs use only 5 estimate values, 90% of the jobs use 20 estimates values etc. Thus, there is no way around the need to develop an accurate model, as is done next.

5.3 Methodology and Roadmap

The modal nature of estimates motivates the following methodology. When examining a trace, we view its estimate distribution as a series of K modes given by $\{(t_i, p_i)\}_{i=1}^{K}$. Each pair (t_i, p_i) represents one mode, such that t_i is the estimate-value in seconds (t for time), and p_i is the percentage of jobs that use t_i as their estimate (p for percent or popularity). For example, the CTC mode series includes the pair $(18h, 23.8\%)$ because 23.8% of the jobs have used 18 hours as their estimate. Occasionally, we refer to modes as *bins* within the estimate histogram. Note that $\sum_{i=1}^{K} p_i = 100\%$ (we are considering all the jobs in the trace).

The remainder of this section serves as a roadmap of this chapter, describing step-by-step how the $\{(t_i, p_i)\}_{i=1}^{K}$ mode-series is constructed, while outlining our methodology. Each of the following paragraphs correspond to a section or two in this chapter, and may contain some associated definitions to be used later on.

Trace Files We build our model carefully, one component at a time, in order to achieve the desired effect. Each step is based on analyzing user estimates in traces from various production machines, in an attempt to find invariants that are not unique to a single installation. To this end we had to apply some manipulations to some of the traces files we use. This is discussed in Section 5.5.

Mass Disparity Our first step is showing that the modes composing the mode-series naturally divide into two groups: About 20 "head" estimate values are used throughout the entire trace by about 90% of the jobs. The rest of the estimates are considered "tail" values. This subject is titled "mass disparity" [41] and is discussed in Section 5.6. We will see that the two mode groups have distinctive characteristics and actually require a separate model. Naturally, the efforts we invest in modeling the two are proportional to the mass they entail.

Number of Estimates We start the modeling in Section 5.7 by finding out how many different estimates there are, that is, modeling the value of K. Note that this mostly effects the tail as we already know the head size (∼20).

Time Ranks The next step is modeling the values themselves, that is, what exactly are the K time-values $\{t_i\}_{i=1}^{K}$. The indexing of this ascendingly sorted series is according to the values, with t_1 being the shortest and t_K being the maximal value allowed within the trace (also denoted E_{max}). The index i denotes the *time rank* of estimate t_i. This concept proved to be very helpful in our modeling efforts. We also define the *normalized time* of an estimate t_i to be t_i/E_{max} (a value between 0 and 1). Section 5.8 defines the function F_{tim} that gets i as input (time rank), and returns t_i (seconds).

Popularity Ranks Likewise, we need to model the mode sizes / popularities / percentages: $\{p_j\}_{j=1}^{K}$. This series is sorted in order of decreasing popularity, so p_1 is the percentage of jobs associated with the most popular estimate. The index j denotes the *popularity rank* of the mode to which p_j belongs. For example, the popularity rank of 18h within CTC is 1 (p_1 = 23.8%), as this is the most popular estimate. We also define the *normalized popularity rank* to be j/K (a value between 0 and 1). Section 5.9 defines the function F_{pop} that gets j as input (popularity rank), and returns p_j, the associated mode size.

Mapping Given the above two series, we need to generate a mapping between them, namely, to determine the popularity p_j of any given estimate t_i, which are paired to form a mode. Section 5.10 defines the function F_{map} that gets i as input (time rank) and returns j as output (popularity rank).

Using the two functions defined above, we can now associate each t_i with the appropriate p_j. This yields a complete description of the estimates distribution. The model is then briefly surveyed in Section 5.11.

Validation Finally, the last part of this chapter is validating that the resulting distribution resembles the reality. Additionally, we also verify through simulation that the "badness" of user estimates is successfully captured, by replacing the original estimates with those generated by our model. The replacement activity mandates developing a method according to which estimates are assigned to jobs (recall that an estimate of a job must be bigger than or equal to its runtime). This is done in Section 5.12.

5.4 Input, Output, and Availability

As we go along, the number of *model parameters* accumulates to around a dozen. Most are optional and are supplied with reasonable default values. The only mandatory parameters are the number of jobs N (the number of estimates to produce), and the maximal allowed estimate value E_{max}. Another important parameter is the percentage of jobs associated with E_{max}, as this popular mode exhibits great variance and has decisive effect on performance. The *output of the model* is the series of the modes: how many jobs use which estimate.

The model we develop is somewhat sophisticated and involves several technical issues with subtle nature. As it is our purpose to allow simulations that are more realistic, the C++ source code of the model is made available for download from the Parallel Workload Archive [110, 155]. Its interface includes of two functions: The first gets a structure containing all the model parameters (all but two are assigned default values), and returns an array of K modes. The second gets the mode array, and another array composed of job structures (which includes ID and runtime). It then associates each job with a suitable estimate, under the constraint that runtime mustn't be bigger. An accompanying shell utility can read SWF file and appending estimates to it.

5.5 Trace Files Manipulation

The analysis and simulations reported in this chapter are, as usual, based on the four accounting logs we have used throughout this entire dissertation. However, during the development of the estimate model we found that SDSC and KTH need to be manipulated, in order to be useful in the context of this chapter. The two "new" trace files are listed in Tab. 5.1, alongside the "old" four, to provide convenient reference. SDSC-106 and KTH4H are the manipulated versions of SDSC and KTH, respectively, to be described next.

SDSC-106 We say an estimate mode is "owned" by a user if this estimate was exclusively used by only that user within the log. It turns out that user 106 is uniquely "creative" in comparison to others, owning 204 estimates of the 543 found in SDSC (nearly 40%). This is highly irregular,[2] as shown in Fig. 5.1, which displays the number of modes owned by each user (only mode owners are shown). We therefore remove this unique activity from the log for the remainder of the discussion (regular activity of user 106, using estimates that are also used by others, is allowed to remain).

[2] In fact, as this activity is concentrated within about 2 months of the log, it actually constitutes a workload flurry as will be defined in Chapter 6.

Abbrev.	Site	Start		End		CPUs	Number of jobs (N)			M	U	X	K
							original	cleaned	sane	*mon*	*usr*	*max*	*est*
SDSC-106	San-Diego SC Ctr.	Apr	98	Apr	00	128	73,103	59,332	53,673	24	428	18h	339
CTC	Cornell Theory Ctr.	Jun	96	May	97	512	79,302	77,222	77,222	11	679	18h	265
KTH4H	Swedish Royal Instit.	Sep	96	Aug	97	100	23,070	23,070	23,070	11	209	4h	106
BLUE	San-Diego SC Ctr.	Apr	00	Jun	03	1,152	250,440	243,314	223,407	32	468	36h	525
SDSC	San-Diego SC Ctr.	Apr	98	Apr	00	128	73,496	59,725	54,053	24	428	18h	543
KTH	Swedish Royal Instit.	Sep	96	Aug	97	100	28,490	28,490	28,490	11	214	60h	271

Table 5.1: *Adding SDSC-106 and KTH4H to the 4 trace files we have used up till now. The variables M, U, X, and K are months duration, number of users, maximal estimate value, and number of estimate bins, respectively. Note that while BLUE's maximal estimate is 36h, its "effective" E_{max} is actually 2h, the limit associated with the "express" and "interactive" queues, used by most jobs within BLUE (Fig. 1.14, page 22).*

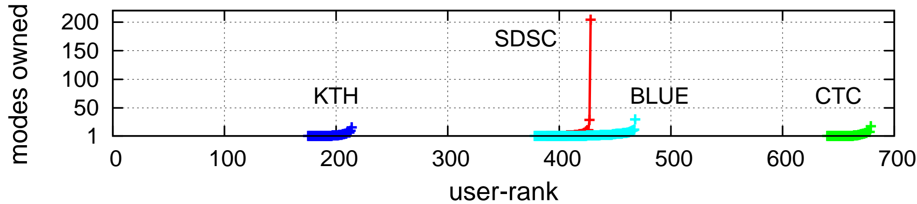


Figure 5.1: *Assume there are n users in a log. Users are associated with the number of modes they own m_i ($i = 1, ..., n$) such that m_1 is the smallest and m_n is the biggest. The index i is defined to be the "user-rank" and serves as an X-value; m_i serves as the associated Y-value. Only positive m_i-s are displayed (users that own no modes are not shown). The SDSC outlier is associated with user 106 which is order of magnitude more "industrious" than other users, exclusively owning 38% of SDSC's modes.*

The resulting log is called *SDSC-106*. This version is beneficial when modeling K in Section 5.7 (number of estimate modes) and F_{tim} in Section 5.8 (actual estimate time values). Other aspects of the model are not affected.

KTH4H The other problematic workload was KTH: This log is actually a combination of three different modes of activity (see bottom of Fig. 1.14, page 22): running jobs of up to 4 hours on weekdays, running jobs of up to 15 hours on weeknights, and running jobs of up to 60 hours on weekends. We have found that in the context of user estimates modeling, considering these three domains in an aggregated manner is similar to, say, aggregating CTC and BLUE to be a single log. We therefore focused on only one of them — the daytime workload with the 4-hour limit, which is the largest component of the log. This is denoted by *KTH4H*.

5.6 Mass Disparity of Estimates

Examining the histogram of estimates immediately reveals that the distribution is highly modal (Fig. 1.14, page 22): A small number of values are used very many times, while many other values are only used a small number of times. In this section, we establish the mass disparity among estimate bins.

Human beings tend to estimate runtime with "round" or "canonical" numbers: 15 minutes, one hour etc. [108, 15, 93]. This has two consequences. One is that the number of bins in the histogram

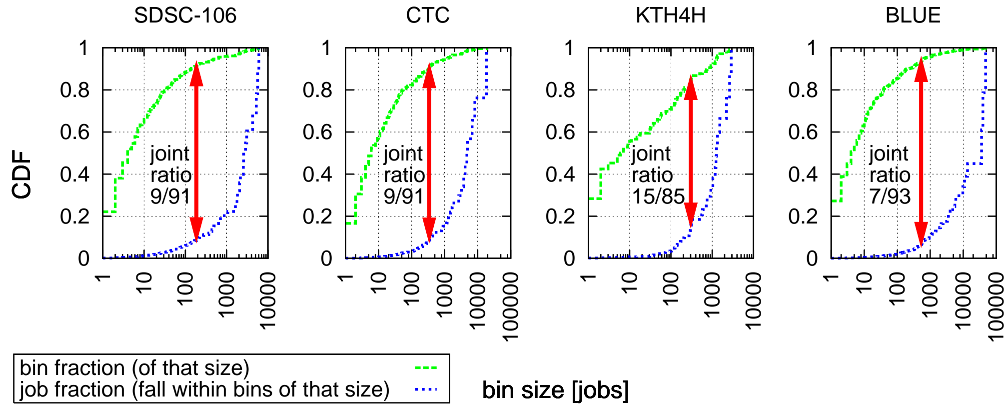


Figure 5.2: *Distributions of bins and of jobs, showing that a small fraction of the bins account for a large fraction of the jobs and vice versa. The actual fractions are indicated by the joint ratio, which is a generalization of the proverbial 10/90 rule.*

(K) is very small relative to the number of jobs in the trace (N). According to Table 5.1, N may be in the order of tens to hundreds of thousands, while K is invariably in the order of only a few hundreds.

The other consequence is that a small set of canonical bins dominates the set of values. Similar phenomena have been observed in many other types of workloads. They are called a "mass disparity", because the mass of the distribution is not spread out equally; rather, a small set of values gets a disproportionally large part of the mass [26].

The mass disparity of user runtime estimates is illustrated in Fig. 5.2. These are CDFs related to the bin size (the number of jobs composing a bin). In each graph, the top line is simply the distribution of bin sizes. This line grows sharply at the beginning, indicating that there are very many small bins (i.e. values that are used by only a small number of jobs). The other line is the distribution of jobs, showing the fraction of jobs with estimates that fall into bins of the different sizes. This line starts out flat and only grows sharply at the end, indicating that most jobs belong to large bins (i.e. most estimate values are the popular values that are repeatedly used very many times).

The figure also shows the joint ratio for each case [41]. This is a generalization of the well-know 10/90 rule. For example, the joint ratio of 9/91 for the CTC log means that 9% of the bins account for 91% of the jobs, and vice versa: the other 91% of the bins contain only 9% of the jobs. Further details about the shape of the distributions are given in Table 5.2. This shows the absolute number of bins involved, rather than their fraction; for example, the CTC row shows that a mere 4 bins cover 50% of the jobs, 10 bins cover 75% of the jobs, and 22 bins contain 90%. Indeed, a bit more than 20 head bins are enough to account for 90% of the jobs in all four logs.

"Head" bins dramatically vary in size: While the most popular is used by $10-27\%$ of the jobs, only $\approx 1\%$ use the 20-th most popular. Regardless, all head bins, whether large or small, have a common temporal quality: their use is not confined to a limited period of time. Rather, they are uniformly used throughout the entire log. This is shown in Fig. 5.3 that plots the number of weeks in which estimates are used, as a function of their popularity ranks. The horizontal dot sequence associated with head bins indicates they are spread out evenly throughout the log. Further, the point of intersection between this sequence and the Y-axis is always the duration of the trace, e.g.

jobs	10%	50%	75%	90%	95%	98%	99%	100%
SDSC-106	1	6	12	22	39	77	116	339
CTC	1	4	10	22	36	62	89	265
KTH4H	1	6	12	21	28	36	43	106
BLUE	1	3	8	23	42	76	116	563
SDSC	1	6	12	23	43	91	156	543
KTH	1	8	21	41	60	89	122	270

Table 5.2: *Mass disparity: per-log minimal number of estimate bins needed to cover the specified percent of the jobs, as noted in the first row.*

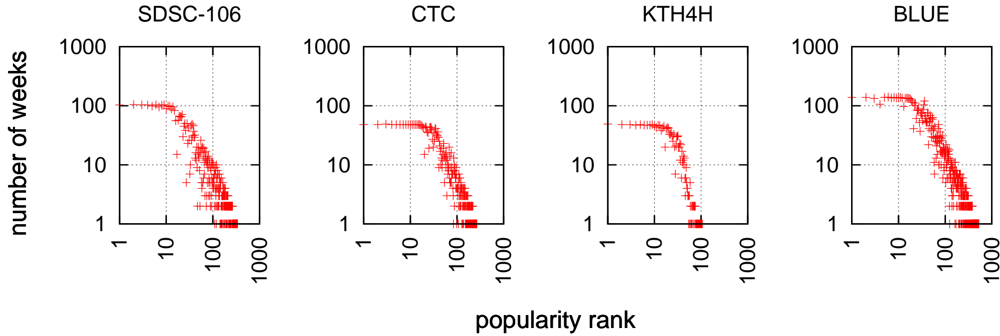


Figure 5.3: *Weeks in which an estimate appears, as a function of its popularity-rank. Recall that using popularity-ranks implies estimates are sorted on the X-axis from the most popular to the least. The top-20 most popular estimates appear throughout the entire logs.*

for SDSC this is 2 years (a bit more than 100 weeks).

5.7 Number of Estimates

We have established that about 20 popular "head" bins represent about 90% of the jobs' estimate distribution mass. We are left with the question of modeling the number of the other "tail" bins used by the remaining 10%.

Examining the four traces of choice in Table 5.1, we see that K tends to grow with the size of the trace, where this "size" can be measured in various ways: as the number of jobs executed (N), as the duration of time spanned (M), as the maximal estimate (X), or as the number of different active users (U). Note that the U metric also measures size, as new users continue to appear throughout each log. This is relevant because after all, users are the ones generating the estimates. In fact, in each of the four traces of choice, about 40% of the estimate modes are exclusively owned (as defined above) by various users.[3]

We have experimented in modeling K as a function of the aspects mentioned above (individually or combined), and most attempts revealed some insightful observations. In fact, we are convinced K is the product of a combination of all factors, and that they all effect it to some degree. However, in the interest of being short while avoiding unwarranted complications (considering this only affects the tail of the distribution), we have chosen to model K as a function of N alone, which obtains tolerable results.

[3]A surprising anecdote is that the actual number of bin-owners is also (exactly) 40, in three of the four traces.

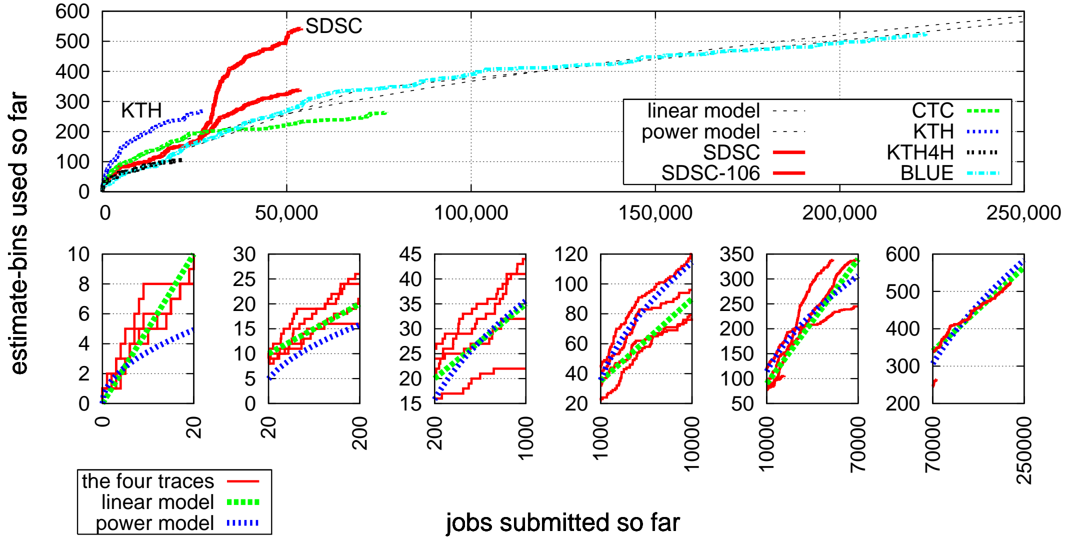


Figure 5.4: *Modeling K using a power model $K = \alpha N^{\beta}$ ($\alpha = 1.1$, $\beta = 0.5$) and a liner model which is defined by the points as specified in Table 5.3. In the top figure, curves associated with SDSC share the same texture (color), the lower is of SDSC-106.*

N (jobs)	0	20	200	1,000	10,0000	70,000	250,000
K (ests)	0	10	20	35	90	340	565
K/N (slope)		1/2	1/18	1/53	1/164	1/240	1/800

Table 5.3: *Points defining the linear model of K using N. Slope indicates the arrival rate of new estimates.*

Fig. 5.4 plots K as a function of the number of jobs submitted so far (if n is an X value, its associated Y is the number of estimate bins in use, before the n-th job was submitted). Note how the vanilla version of KTH and SDSC stands out: the former due to the three estimate domains it contains, and the latter due to user 106. All curves can be rather successfully fitted with a power model on individual bases (we present one such power model that was simultaneously fitted against all four traces of choice).

Accordingly, we allow the user of our model to supply the appropriate coefficients (as optional parameters). However, as this only effects tail bins, we set an ad-hoc linear model (defined by Table 5.3) as the default configuration. This provides a tolerable approximation of K for any given job number N.

5.8 Time Values of Estimates

Having computed a K approximation (order of a few hundreds), we know how many estimate bins should be produced by our model. Let us continue to generate these K values, namely manufacture the $\{t_i\}_{i=1}^{K}$ series. It has already been noted that users tend to give "round" estimates [108, 17, 93], but this loose specification is not enough. In this section we develop a simple method to generate K such appropriate values. We are currently not considering the most popular (20) estimates in a separate manner. These will be addressed in detail later on (Section 5.10), complementing the model we develop in this section.

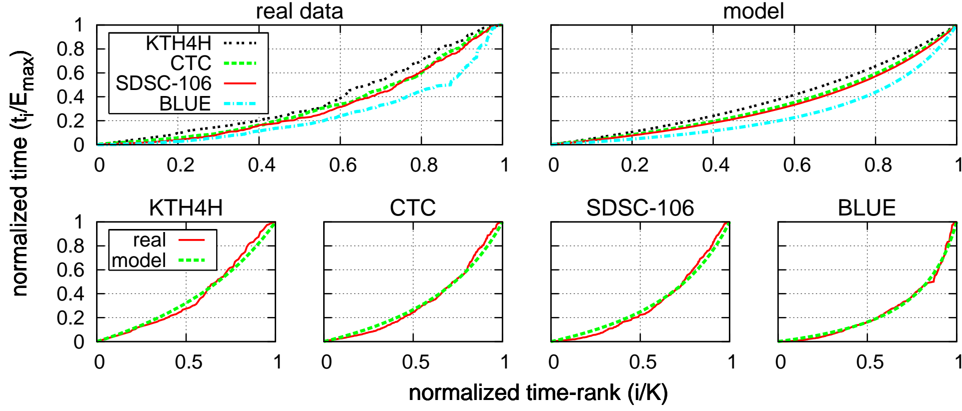


Figure 5.5: *Modeling estimate times using* $f(x) = \frac{(a-1)x}{a-x}$.

trace	KTH4H		CTC		SDSC-106		BLUE
a	1.91	$>$	1.57	$>$	1.50	$>$	1.24
K	106	$<$	265	$<$	339	$<$	525

Table 5.4: *The a parameter of the fractional fit presented in Fig. 5.5 is correlated with the number of different estimates (K).*

Recall that the time-ranks of estimates are their associated *indexes*, when ascendingly numbered from shortest to longest. Evidently, this concept can be very helpful for our purposes. We define a function F_{tim} that upon a time-rank input i, returns the associated time value t_i (seconds), such that $F_{tim}(i) = t_i$.

The top-left of Fig. 5.5 plots normalized estimate time (t_i/E_{max}, where E_{max} is the maximal estimate) as a function of its associated normalized time-rank (i/K), for all four traces. According to the top-right and bottom of Fig. 5.5, it turns out the resulting curves can be modeled with great success when using the fractional function $f(x) = \frac{(a-1)x}{a-x}$ for some $a > 1$ (x is normalized time-rank). Further, the actual values of a (Table 5.4) are negatively correlated with K, in that bigger K implies smaller a.

An obvious property of $f(x)$ in the relevant domain ($x \in [0,1]$) is that when a gets closer to 1, its numerator goes to zero and therefore the curve gets closer to the bottom and right axes. On the other hand, as a gets further from 1 (goes to infinity), its numerator and denominator get more and more similar, which means the function converges to $f(x) = x$ (the main diagonal). The practical meaning of this is that less estimate values (smaller K, bigger a) means estimates' temporal spread is more uniform. In contrast, more estimate values (bigger K, smaller a) means a tendency of estimates to concentrate at the beginning of the Y-axis, namely, be shorter.

In order to reduce the number of user-supplied parameters of our model, we can approximate a as a function of K (which we already know how to reasonably deduce from the number of jobs). The problem is that we only have four samples (Table 5.4), too few to produce a fit. One heuristic to overcome this problem is splitting the traces in two and computing K and a for each half. This enlarges our sample space by eight (two additional samples per trace) to a total of twelve. The results of fitting this data to the best model we could find are shown in Fig. 5.6 and indicate a moderate success.

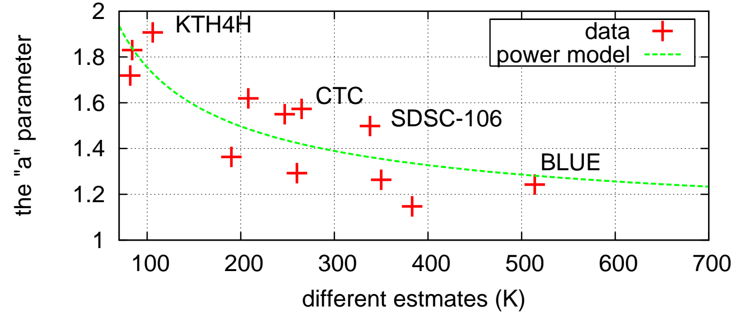


Figure 5.6: *Modeling a as a function of K using $1+\alpha K^{\beta}$ (with $\alpha = 12.1$, $\beta = -0.6$). A bigger K results in an a-parameter that is closer (but never equal) to 1, as required.*

We can now define the required F_{tim} to be

$$F_{tim}(i) = E_{max} \cdot f\,(i/K) = E_{max} \cdot \frac{(a-1)\frac{i}{K}}{a - \frac{i}{K}} \quad .$$

Generating the $\{t_i\}_{i=1}^{K}$ series of time values is done by simply assigning $1, 2, ..., K$ to the time-rank i in an iterative manner. Finally, as almost 100% of the estimates are given in a minute resolution, the generated values are rounded to the nearest multiple of 60 (if not colliding with previously generated estimates).

5.9 Popularity of Estimates

In the previous section we have modeled the time values of estimates. Here we raise the question of how popular is each estimate, that is, how many jobs actually use each estimate value? Answering this question implies modeling the $\{p_i\}_{i=1}^{K}$ percentage series. Once again, like in the previous section, ranking the estimates (this time based on popularity) proves to be highly beneficial. Recall that $\{p_i\}_{i=1}^{K}$ is descendingly sorted such that p_1 is the percentage of jobs using the most popular estimate value, p_i is the percentage of jobs using the i-most popular estimate value, and i serves as the associated popularity rank. We seek a function F_{pop} such that $F_{pop}(i) = p_i$. Note that the constraint of $\sum_{i=1}^{K} F_{pop}(i) = 100$ must hold.

Fig. 5.7 plots the size (percent) of each estimate bin, as a function of its popularity-rank. There's a clear distinction between the top-20 most popular estimates (distribution's head) and the others (tail), in that the sizes of head-bins decay exponentially, whereas the decay of the tail obeys some power law.

The suggested fits are indeed very successful ($R^2 > 0.95$ in both cases). However, when concentrating on the head (left or middle of Fig. 5.7), it is evident the exponential model is less successful for the first few estimates. For example, in CTC the most popular estimate is used by about 24% of the jobs, while in SDSC this is true for only 11%. In BLUE the situation is worse as the three top ranking estimates "break" the exponential curve. (Indeed, the exponential fit was produced after excluding these "abnormal" points.) Obviously, no model is perfect. But this seemingly minor deficiency (at the "head of the head") is actually quite significant, as a large

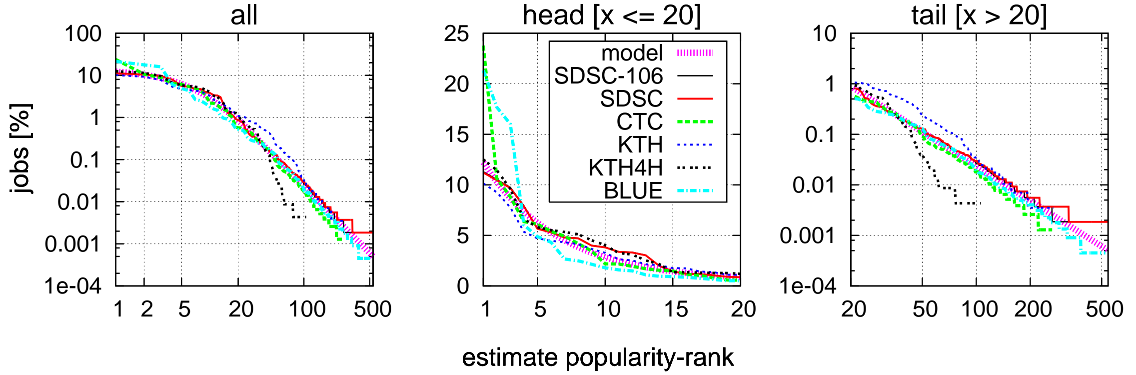


Figure 5.7: *Modeling percent of jobs associated with estimate bins, as a function of popularity rank. The head (middle) is modeled by the exponential function $\alpha e^{\beta x} + \gamma$ (with $\alpha = 14.05$, $\beta = -0.18$, and $\gamma = 0.46$). The tail (right) is modeled by the ωx^{ρ} power law (with $\omega = 795.6$ and $\rho = -2.27$). Note that the middle figure has linear axes, while the other two are log scaled. The left figure concatenates the head and tail models.*

part of the distribution mass lies within this part (differences in less popular estimates are far less important).

We note that the observed differences among the traces at the “head of the head” expose an inherent weakness in any estimate model one might suggest, because the effect of the variance among these 1-3 estimates is decisive. Consequently, our model will allow (though not mandate) the user to provide information regarding top-ranking estimates as model parameters (this will be further addressed in the next section). As for the default, recall that a job estimating to run for the maximal allowed value (E_{max}) is the worst kind of job in the eyes of a backfilling scheduler (Section 5.2). For this reason, we prefer the default model to follow the CTC example by making the (single) top ranking estimate “break” the exponential contiguity. This significant job percentage will later be associated with E_{max} to serve as a realistic worst case scenario. We therefore define F_{pop} as follows

$$F_{pop}(i) = \begin{cases} 89 - \sum_{j=2}^{20} \left(\alpha e^{\beta \cdot j} + \gamma\right) & i = 1 \\ \alpha e^{\beta \cdot i} + \gamma & i = 2, 3, ..., 20 \\ \omega \cdot i^{\rho} \cdot \frac{100-89}{\lambda} & i = 21, 22, ..., K \end{cases}$$

Starting with the (simplest) middle branch, F_{pop} is determined by the exponential model for all head popularity ranks but the first (the default values for the coefficients are specified in the caption of Fig. 5.7). The first branch is defined so as to preserve the invariant shown in Table 5.2 that the twenty top ranking estimates are enough to cover almost 90% of the jobs. Finally, the third branch determine sizes of tail estimates according to a power law (again, coefficient values are specified in Fig. 5.7). But to preserve the constraint that $\sum_{i=1}^{K} F_{pop}(i) = 100$, tail sizes are scaled by a factor of $\frac{100-89}{\lambda}$, where λ is the sum of the tail: $\sum_{i=21}^{K} \omega \cdot i^{\rho}$. The resulting default curve is almost identical to the one associated with the model as presented in Fig. 5.7, with a top rank of a bit more than 20% (to be associated with E_{max}).

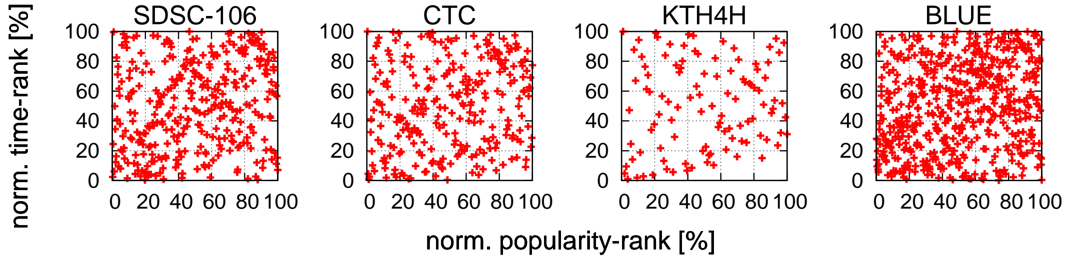


Figure 5.8: *Scatter plots of relative popularity-ranks vs. relative time-ranks appear to reveal a uniform distribution across all traces.*

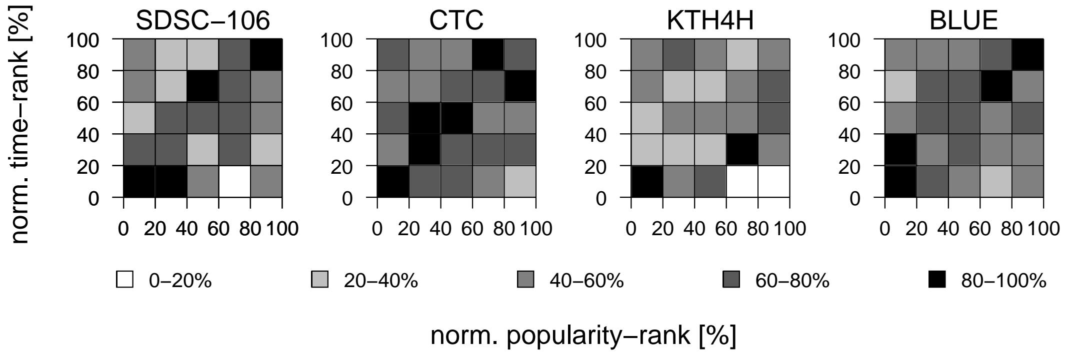


Figure 5.9: *Aggregating the data shown in Fig. 5.8 into a grid-based heat-map reveals no further insight, other than a consistent tendency of popular estimates to be short (bottom-left black cells).*

5.10 Mapping Time to Popularity

The next step after separately generating the estimates' time $\{t_i\}_{i=1}^{K}$ and popularity $\{p_j\}_{j=1}^{K}$ is figuring out how two construct a bipartite matching between the two. We seek a function F_{map} such that $F_{map}(i) = j$, that is, we want to map each time-rank to a popularity-rank in a manner that yields estimate distributions similar to those found in the original traces (Fig. 1.14, page 22).

5.10.1 Mapping of Tail Estimates

As a first step towards constructing F_{map}, let us examine this mapping as it appears in the four traces. Fig. 5.8 scatter plots normalized popularity-ranks vs. normalized time-ranks: one point per estimate.[4] The points appear to be more or less uniformly distributed, which means there is no apparent mapping rule.

In an effort to expose some trend possibly hidden within the "disorder" of the scatter plots, we counted the number of points in each grid-cell within Fig. 5.8. We then generated an associated heat-map for each sub-figure by assigning a color based on the point-count of each cell: cells that are populated by 80-100% of the maximal (cell) point-count found within the sub-figure (denoted C), are assigned with black; cells populated by 0-20% of C are assigned with white; the remaining cells are assigned with a gray intensity that is linearly proportional to their point-count, batched in multiples of 20% of C.

[4]A scatter plot of actual values turns out to be meaningless.

The result, displayed in Fig. 5.9, appears to strengthen our initial hypothesis that the mapping between popularity-ranks and time-ranks is more or less uniformly random, as other than the bottom-left cell being consistently black (top-20 popular estimates show tendency of being shorter), there is no consistent pattern that emerges when comparing the different traces.

Our next step was therefore to randomly map between time and popularity ranks. Regrettably, this resulted in failure, as the generated CDFs were significantly different than those displayed in Fig. 1.14 (page 22), because "big modes" fell in wrong places. The fact of the matter is that when (uniformly) randomly mapping between time and popularity ranks, there is a nonnegligible probability that the 4-5 most popular estimates are assigned to (say) times in the proximity of the maximal value, which means that the majority of the distribution mass is much too long. Alternatively, there is also a nonnegligible probability that the opposite will occur, namely, that none of the more popular estimates will be assigned to a time in the proximity of E_{max}, contrary to our previous findings.

We conclude that it is tail estimates (in terms of popularity) that are roughly randomly mapped to times in a uniform manner, forming the relatively balanced scatter plot observed in Fig. 5.8. This appearance is created due to the fact there are much more tail estimates (few hundreds) than head's (20). The head estimates minority, which nevertheless constitute 90% of the mass, distributes differently and requires a greater modeling effort.

5.10.2 Determining Head Times

We have reached the point where the effort to model user estimates is reduced to simply determining 20 actual time-values and mapping them correctly to the appropriate (head) sizes. In other words, our task is as simple as producing 20 (t_i, p_i) pairs. These are good news, as the number of samples is small enough to allow a thorough examination of the entire sample-space. The bad news is that unlike previous parts of the model that are actually relatively trivial, and in spite of considerable efforts we've made, we failed to produce a *simple* method to accomplish the task. In the interest of practicality and space, we do not describe our various unsuccessful attempts to produce a simple straightforward solution. Instead, we concentrate on describing the sophisticated algorithm we've developed that has finally managed to deliver satisfactory results.

Let us examine the relevant sample space. Table 5.5 lists the 20 most popular estimates in each trace, and their associated sizes (percent of jobs). Of the 36 values displayed, a remarkable 15 are *joint times* across all traces (we ignore KTH4H when deciding which values, bigger than 4h, are joint). The joint times are highlighted in bold font, and have values one would expect from humans to ordinarily use. Note that this is regardless of the different per-trace maximal estimate limits. We conclude that joint times should be hard-coded in our model, as it is fairly reasonable to conjecture humans will always extensively use values like 15 minutes, 1 hour, etc. We therefore define the first head-mapping step — determining the 20 time values that are the most popular — as follows:

1. Choose E_{max}, the maximal estimate (which is a mandatory parameter of our model). As previously mentioned, this is always a top ranking value.
2. Choose all hard-coded joint times that are smaller than E_{max}.
3. Choose in descending order multiples of T_{round} (smaller than E_{max}), where T_{round} is 200h,

#	estimate *hh:mm*	SDSC-106	CTC	KTH4H	BLUE
1	00:01			6.6 (4)	
2	00:02			4.0 (10)	
3	00:03			2.2 (14)	
4	00:04			1.2 (20)	
5	**00:05**	11.3 (1)	8.8 (3)	11.5 (2)	2.7 (7)
6	**00:10**	7.9 (4)	6.4 (4)	9.6 (3)	4.3 (6)
7	00:12	1.2 (17)			
8	**00:15**	3.0 (13)	10.6 (2)	5.3 (7)	16.0 (3)
9	**00:20**	4.8 (7)	2.0 (12)	3.1 (12)	2.5 (8)
10	**00:30**	4.7 (8)	3.5 (9)	5.5 (6)	17.7 (2)
11	00:40			1.3 (19)	0.5 (19)
12	00:45	1.1 (18)			
13	00:50				0.5 (20)
14	**01:00**	10.5 (2)	4.2 (8)	5.8 (5)	4.9 (5)
15	01:30		0.8 (18)	1.3 (18)	1.5 (12)
16	01:40			1.4 (16)	
17	01:59				6.0 (4)
18	**02:00**	5.3 (6)	5.4 (6)	4.5 (9)	21.3 (1)
19	02:10			1.3 (17)	
20	02:30	1.2 (16)		1.4 (15)	
21	**03:00**	3.8 (10)	4.9 (7)	2.5 (13)	1.8 (10)
22	03:20			5.1 (8)	
23	03:50			3.3 (11)	
24	**04:00**	5.7 (5)	2.2 (11)	12.5 (1)	1.6 (11)
25	04:50		0.6 (20)		
26	**05:00**	1.4 (15)	1.1 (16)		0.9 (15)
27	**06:00**	2.0 (14)	6.1 (5)		1.0 (14)
28	07:00	0.9 (19)			
29	**08:00**	3.4 (11)	1.5 (14)		0.8 (17)
30	**10:00**	3.3 (12)	1.7 (13)		0.9 (16)
31	**12:00**	4.0 (9)	2.2 (10)		0.6 (18)
32	15:00	0.9 (20)	1.5 (15)		
33	16:00		1.0 (17)		
34	17:00		0.6 (19)		
35	**18:00**	9.8 (3)	23.8 (1)		2.1 (9)
36	36:00				1.1 (13)
sum (all)		86.4	88.9	89.3	88.7
sum (joint)		81.2	84.4	60.4	79.1

Table 5.5: *The top-20 most popular modes in the four traces. Each column contains exactly 20 job percent values. Note that 15 of the top-20 estimates are joint across all traces (excluding KTH4H for estimates bigger than 4 hours). Joint estimates are highlighted in bold font. The parenthesized subscripts denote the associated popularity-ranks (e.g. in BLUE, 2h is the most popular value used by* 21.3% *of the jobs). Notice that the sum of each column is invariantly in the neighborhood of 89%, the value we used in Section 5.9 to define* F_{pop}.

then 100h, 50h, 10h, 5h, 2h, 1h, 20m, 10m, and 5m. We stop when the number of (different) chosen values reaches 20.

The role of the third item above is to add a *relative* aspect to the process of choosing popular estimates, which is largely hard-coded. As will later be shown, this manages to successfully capture KTH4H's condensed nature. At the other end, workloads with larger estimate domains, of jobs that span hundreds of hours, do in fact exist [17]. Regrettably, their owners refuse to share them with the community. Nevertheless, our algorithm generates longer times based on the modes they report (400h, 200h, 100h, and 50h in the NCSA O2K traces).

Finally, recall we have already generated K time values using F_{tim} defined in Section 5.8. Head times generated here, replace the 20 values generated by F_{tim} that are the closest to them (and so the structure reported in Fig. 5.5 is preserved).

5.10.3 Mapping of Head Estimates

Having both head times (seconds) and sizes (job percentages), we go on to map between them. As usual, mapping is made possible by using the associated ranks, rather than the actual values. For this purpose we need two new definitions:

First, we define a new type of time-rank, the *top-20 time rank* (or *ttr* for short), which is rather similar to the ordinary time-rank: All top-20 times, excluding E_{max}, are ascendingly sorted. The first is assigned a ttr=1, the second a ttr=2, and the last a ttr=19. For example, according to Table 5.5, in CTC, 00:05 has ttr=1, 00:10 has ttr=2, 01:30 has a ttr=7, and 17:00 has a ttr=19. E_{max} is always associated with ttr=0.

Second, for each trace-file log, we define a function F_{log} that maps ttr-s to the associated popularity ranks, within that log. For example, $F_{ctc}(0)$=1 as E_{max}=18h (associated with ttr=0) is its most popular estimate. Likewise, $F_{ctc}(1)$=3, as 5min is the smallest top-20 estimate (ttr=1) and is the third most popular estimate within CTC. Table 5.6 lists F_{log} of the four traces. Recall that 2h is the effective E_{max} of BLUE and therefore this is the estimate we choose to associate with ttr=0. Additionally, note the BLUE 01:59 mode near its E_{max}=2h (Table 5.5). This is probably due to users trying to enjoy both worlds: use the maximal value, while "tricking" the system to assign their jobs a higher priority as a result of being shorter. We are not interested (nor able) to model such phenomena. Therefore, in the generation of Table 5.6 and throughout the reminder of this chapter, we aggregate the 01:59 mode with that of 2h and consider them a single 27.3% mode.

The F_{log} functions in Table 5.6 reflect reality, and are in fact the reason for the log-uniform CDFs observed in Fig. 1.14 (page 22). We therefore seek an algorithm that can "learn" these functions and be able to imitate them. Given such an artificial F_{log}, we would finally be able to match head-sizes (produced in Section 5.9, their size defines their popularity rank) to head-times (produced in Section 5.10.2, their value defines their ttr-s) and complete our model.

At first glance, the four F_{log} functions appear to have little similarities (the correlation coefficient between the columns of Table 5.6 is only 0.1-0.3), seemingly deeming failure on the generalization attempt. However a closer inspection reveals some regularities. Consider for example the more popular (and therefore more important to model) ranks: at least three of four values of each such rank are clustering across neighboring lines (ttr-s). This is made clearer in Fig. 5.10.

Another observation is that when dividing popularity-ranks into two (1-10 vs. 11-20), around 75% of the more popular ranks are found in the top half of Table 5.6, which indicates a clear

ttr	$F_{sdsc-106}$	F_{ctc}	F_{kth4h}	F_{blue}
0	**3**	**1**	**1**	**1**
1	**1**	**3**	**4**	6
2	**4**	**4**	10	5
3	17	**2**	14	**3**
4	13	12	20	7
5	7	9	**2**	**2**
6	8	8	**3**	18
7	18	18	7	19
8	**2**	6	12	**4**
9	6	7	6	11
10	16	11	19	20
11	10	20	5	9
12	5	16	18	10
13	15	5	16	14
14	14	14	9	13
15	19	13	17	16
16	11	10	15	15
17	12	15	13	17
18	9	17	8	8
19	20	19	11	12

Table 5.6: *The F_{log} functions of the four traces. The four most popular ranks in each trace are highlighted in bold font.*

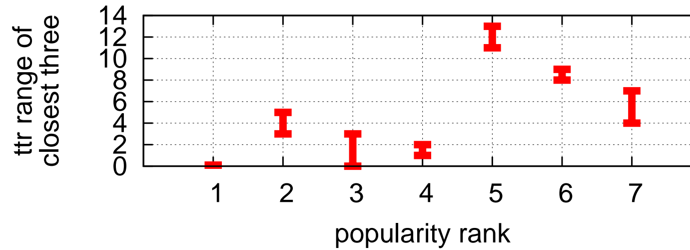


Figure 5.10: *There is only 0-3 difference between the closest three ttr-s that are associated with the more popular ranks (Table 5.6). For example, 3 of the ttr-s associated with popularity rank 2, are located in rows 3-5 in Table 5.6 (underlined and highlighted in a different color). In the above figure, this corresponds to range-bar associated with popularity rank 2 that stretches between lines 3-5.*

tendency of more popular ranks to be associated with smaller ttr-s. (This coincides with the log-uniformity of the original estimate distributions). It is our job to capture these regularities.

In the initialization part of our algorithm, which we call the *pool algorithm*, we associate ttr=0 (of E_{max}) with popularity rank=1, that is, the maximal estimate is also the most popular. The rationale of this decision is that

1. according to Table 5.6 this is usually the case in real traces,

2. as explained in Section 5.2, making E_{max} the most popular estimate constitutes a realistic worst case scenario, which is most appropriate to serve as the default setting, and

3. it is the "safest" decision due to the constraint that estimates must be longer than runtimes.

The last two items are the reason why we chose to follow the CTC example and enforce a sizable first rank on the construction of F_{pop} (end of Section 5.9) that “breaks” the exponential contiguity observed in Fig. 5.7. To complete the initialization part, we allocate an empty vector V_{pool} designated to hold popularity ranks. Any popularity rank may have up to four occurrences within V_{pool}.

The body of the pool-algorithm iterates through the rest of the ttr-s in ascending order ($J_{ttr} = 1, ..., 19$) and performs the following steps on each iteration:

1. For each trace file log, insert the popularity rank $F_{log}(J_{ttr})$ to V_{pool}, but only if this rank wasn’t already mapped to some smaller ttr in previous iterations. (In other words, insert all the values from within the J_{ttr} line in Table 5.6, that weren’t already chosen.)

2. If there exists popularity ranks that have four occurrences within V_{pool}, choose the smallest of these ranks R, map J_{ttr} to R, remove all occurrences of R from V_{pool}, and move on to the next iteration.

3. Otherwise, randomly choose two (not necessarily different) popularity ranks from within V_{pool}, map the smaller of these to J_{ttr}, and remove all its occurrences from V_{pool}.

A main principle of the algorithm is the gradual iteration over Table 5.6, such that no popularity-rank R is eligible for mapping to J_{ttr}, before we have actually witnessed at least one occasion in which R was mapped to a ttr that is smaller than or equal to J_{ttr}. This aims to imitate the original F_{log} functions, along with serving as the first safety-mechanism obstructing more popular ranks to be mapped to longer estimates (recall that estimate CDFs are log-uniform, which means most estimates are short).

Another important principle of the algorithm is that increased number of occurrences of the same R within V_{pool}, implies a greater chance of R to be randomly chosen. And so, an R that is mapped to a ttr $\leq J_{ttr}$ within two traces (two occurrences within V_{pool}), has double the chance of being chosen in comparison to a popularity rank for which this condition holds with respect to only one trace (one occurrence within V_{pool}). This aspect of the algorithm also aims to capture the commonality between the various traces.

Item number two in the algorithm tries to make sure an R will not be mapped to a ttr that is bigger than *all* the ttr-s to which it was mapped in the four traces. Like the first principle mentioned above, this item has the role of making sure the resulting mapping isn’t too different than that of the original logs. It also serves as the second safety-mechanism limiting the probability of more popular ranks to be mapped to longer estimates.

The combination of the above “safety mechanisms” was usually enough to produce satisfactory results. However, on rare occasions, too many high popularity ranks have managed to nevertheless “escape” these mechanisms and be mapped to longer estimates. Adding a third safety-mechanism, in the form of using the minimum between two choices of popularity ranks (third item of the algorithm), has turned this probability negligible.

5.10.4 Embedding User-Supplied Estimates

While the estimate distributions of the traces bare remarkable resemblance, they are also very distinct within the “head of the head” (as discussed in Section 5.9), that is, the 1-3 most popular

estimates. For example, considering Table 5.5, the difference between the percentage of SDSC and CTC jobs associated with 18h (10% vs. 24%) is enough to yield completely different distributions. Another example is BLUE's shift of the maximum from 36h to 2h, or its two huge modes in 15min and 30min; the fact that more than 60% of its jobs use one of these estimates (along with 01:59), cannot be captured by any general model. Yet another example is KTH4H's unique modes below 5min. This variance among the most important estimate bins, along with the fact users may be aware of special queues and other influential technicalities concerning their site, mandates a general model to allow its user to manually supply head estimates as parameters.

To this end, we allow the user to supply the model with a vector of up to 20 (t_i, p_i) pairs. The manner in which these pairs are embedded within our model is the following: The t_i values replace default-generated head times (Section 5.10.2) that are the closest to them, with the exception of E_{max} which is never replaced unless explicitly given by the user as one of the (t_i, p_i) pairs. (This is due to the reasons discussed in Section 5.10.3.) As an example, in order to effectively replace the maximal value of BLUE, the user must supply two pairs: $(36h, 1\%)$ to prevent the model from making the old maximum (36h) the most popular estimate, and $(2h, 27\%)$ to generate the new maximum.

Similarly to times, user supplied p_i sizes (job percents) replace default-generated sizes (Section 5.9) that are the closest to them. Once again, the biggest value (reserved for E_{max}) is not replaced if the user did not supply a pair containing E_{max}. Additionally, the remaining non-user head-sizes are scaled such that the total mass of the head is still 89% (scaling however does apply to the largest non-user size). If scaling is not possible (sum of user sizes exceed 89%), non-user head-sizes are simply eliminated, and the tail sizes are scaled such that the sum of the entire distribution is 100%.

Finally, the pool algorithm is refined to skip ttr-s that are associated with user-supplied estimates and to avoid mapping their associated popularity ranks.

5.11 Overview of the Model

Now that all the different pieces are in place, let us briefly review the default operation of the estimates model we have developed:

1. Get input. The mandatory parameters are maximal estimate value E_{max}, and number of jobs N (which is the number of estimates the model must produce as output). A third, "semi mandatory", parameter is the percentage of jobs associated with E_{max}. While the model can arbitrarily decide this value by itself, its variation in reality is too big to be captured by a model, whereas its influence on performance results is too detrimental to be ignored (E_{max} jobs are the "worst kind" of jobs in the eyes of the scheduler; Section 5.2).

2. Compute the value of K (different estimate times) as defined in Section 5.7.

3. Generate K time-values using F_{tim} as defined in Section 5.8.

4. Generate 20 "head" time-values using the algorithm defined in Section 5.10.2 and combine them with the K time-values produced in the previous item. Non-head times are denoted "tail" times.

5. Generate K sizes (jobs percent) using F_{pop} as defined in Section 5.9. The largest 20 sizes are the head sizes. The rest are tail.

6. Map between time- and size-values using F_{map} as defined in Section 5.10, by

   - Randomly mapping between tail-times and tail-sizes in a uniform manner (Section 5.10.1).
   - Mapping head-times and head-sizes using the pool algorithm (Section 5.10.3).

7. If received user supplied estimate bins, embed them within the model as described in Section 5.10.4.

5.11.1 About the Complexity

The only part which is non-trivial in our model is the pool algorithm: Generating the estimate time values by themselves is a trivial operation. Generating sizes (percentages of jobs) is equally trivial. Mapping between these two value sets is also a relatively easy operation, as all but the 20 most popular sizes can be randomly mapped. All the complexity of the model concentrates in solving the problem of deciding how many jobs are associated with each "head" estimate, or in other words, where exactly to place the larger modes. The question of whether a simpler alternative than the one suggested here exists, is an open one, and it is conceivable there's a positive answer. However, all the "immediate" heuristics we could think of in order to perform this task in a simpler manner have been checked and verified to be inadequate. In fact, it is these inadequacies that has lead us step by step in the development of the pool algorithm.

5.12 Validating the Model

Having implemented the estimate model, we now go on to validate its effectiveness. This is essentially composed of two parts. The first is obviously making sure that the resulting distribution is similar to that of the traces (Section 5.12.1). However, this is not enough by itself, as our ultimate goal is to allow realistic performance evaluation. The second part is therefore checking whether performance results obtained by using the original data are comparable to those produced when replacing original estimates with artificial values produced by the model (Section 5.12.3). The latter part mandates developing a method according to which artificial estimates are assigned to jobs (Section 5.12.2).

5.12.1 Validating the Distribution

Fig. 5.11 plots the original CDFs (solid line) against those generated by the "vanilla" model using various seeds. The only input parameters that are given to the model are those listed in Section 5.11, that is, the maximal estimate E_{max}, then number of jobs N, and the percentage of jobs associated with E_{max}. Recall that BLUE's maximum is considered to be 2 hours and that in order to reflect this we must explicitly supply the model with an additional pair (Section 5.10.4).

The results indicate the model has notable success in generating distributions that are remarkably similar to that of SDSC-106 and CTC; it is far less successful with respect to the other two traces. However, this should come as no surprise because, as mentioned earlier, the model has no

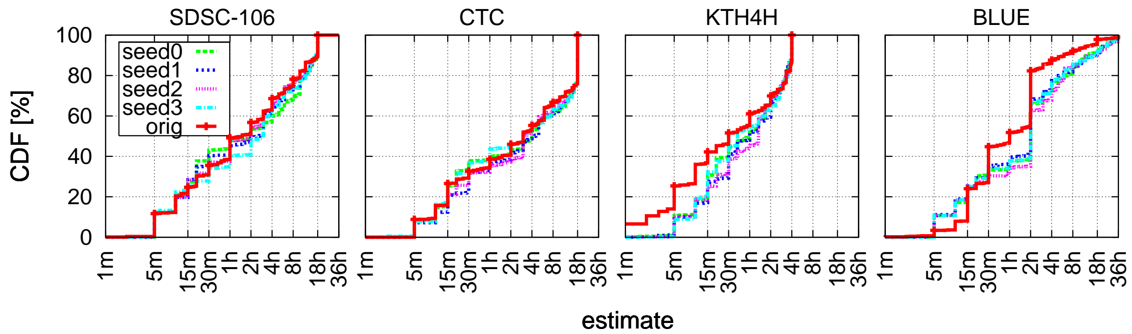


Figure 5.11: *The original estimate distribution of the traces (solid lines) vs. the output of the vanilla model, when used with four different seeds. Output is less successful for traces with unique features.*

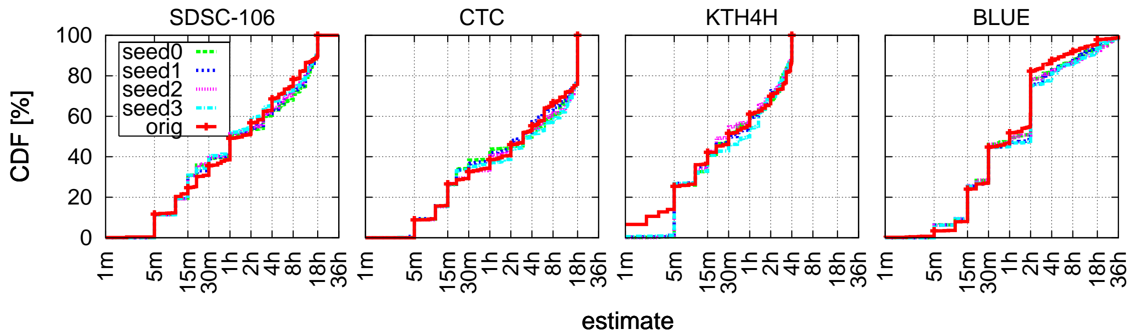


Figure 5.12: *Output of the model under the "improved" setting which provides minimal information identifying the unique features.*

pretense of reflecting abnormalities or features that are unique to individual traces. In the case of KTH4H, these are the large modes that are found below 5 minutes (Table 5.5). In fact, if aggregating these modes with that of 5 minutes, we get that a remarkable 25.5% of KTH4H's jobs have estimates that are 5 minutes or less, which is inherently different in comparison to the other traces. In the case of BLUE, its uniqueness takes the form of two exceptional modes located at 15 and 30 minutes. This distinctive quality is especially apparent in Fig. 5.7, where the three biggest modes "break" the log-uniform contiguity.

The practical question is therefore if the model can produce good results when provided with *minimal* additional information highlighting the trace-specific abnormalities. The amount of such information is inherently limited if we are to keep the model applicable and maintain its practical value. We therefore define the "improved" setting in which the KTH4H model is provided with the additional $(5min, 25\%)$ pair. The BLUE model is provided with two additional pairs associated with its two exceptional modes: $(15min, 16\%)$ and $(30min, 18\%)$.

The results of the improved setting are shown in Fig. 5.12 and indicate that this additional information was all that the model needed in order to produce satisfactory results (also) with respect to the two "unique" traces. To test the impact of additional information on situations where the vanilla model manages to produce reasonable results by itself, the improved setting supplied three additional pairs (of the most popular estimates) when modeling CTC and SDSC-106. It is not apparent whether the additional information made a qualitative difference.

The important conclusion that follows from the successful experiment we have conducted in

this section, is that estimate distributions are indeed extremely similar: Most of their variance concentrates within the 1-3 most popular estimates, and once these are provided, the model produces very good results.

5.12.2 Assigning Estimates to Jobs

The next step in validating the model is putting it to use within a simulation. For this purpose we have decided to simulate the EASY scheduler and evaluate its performance under the four workloads. This can be done with original estimates or after replacing them with artificial values that were generated by our model. Similar performance results would indicate success.

The common practice when modeling a parallel workload is to define canonical random variables to represent the different attributes of the jobs, e.g. runtime, size, inter-arrival time etc. [30, 77, 99]. Generating a workload of N jobs is then performed by creating N samples of these random variables. Importantly, each sample is generated *independently* of other samples.

In this respect, the assignment of artificial estimates to jobs is subtle, as this must be done under the constraint that estimates mustn't be smaller than the runtimes of the jobs to which they are assigned. Here, we can't just simply randomly choose a value. However, if independence between jobs is still assumed, we can easily overcome the problem by using the *random shuffle algorithm*. This algorithm gets two vectors $V_{estimate}$ and $V_{runtime}$ that hold N values as suggested by their names. The content of both vectors is generated as usual, according to the procedure described above (under the assumption of independence). Now all that is needed is a random permutation that maps between the two, such that every estimate is equal to or bigger than its associated runtime. The random shuffle algorithm finds such a permutation as follows. First, it sorts the two vectors; call the sorted versions $SV_{estimate}$ and $SV_{runtime}$. Next, it iterates through $SV_{runtime}$ *from the top down*, i.e. starting with the largest runtimes. For each runtime $SV_{runtime}[i]$, it finds the smallest index j such that $SV_{runtime}[j] \geq SV_{runtime}[i]$. This identifies the legal estimates to use: they are those from that index to the end. The algorithm then picks one of these estimates at random, and pairs is with the ith runtime. After values are paired, they are removed from their respective vectors.

Note that we do not claim that the independence assumption underlying the random shuffle algorithm is correct. On the contrary. We only argue that this is the common practice. However, there is a way to transform the original data such that this assumption holds: The algorithm can be applied to the original data, that is, we can populate the $V_{estimate}$ vector with original trace estimates and reassign them to jobs using the shuffle algorithm. The outcome of doing this would be that the original estimates are "randomly shuffled" between jobs (which is the source of the algorithm's name). The result of such shuffling is to create independent "real" estimates. This is suitable as a basis for comparison with our model, as explained below.

5.12.3 Validating Performance Results

Several estimate-generation models have been evaluated and compared against the original data:

- The *X2*-model: simply doubles user estimates on the fly (as in e.g. Chapter 4).
- The *shfl*-model: the result of applying the random shuffle algorithm (defined above) to the original data. As noted, assuming independence in this context is correct.

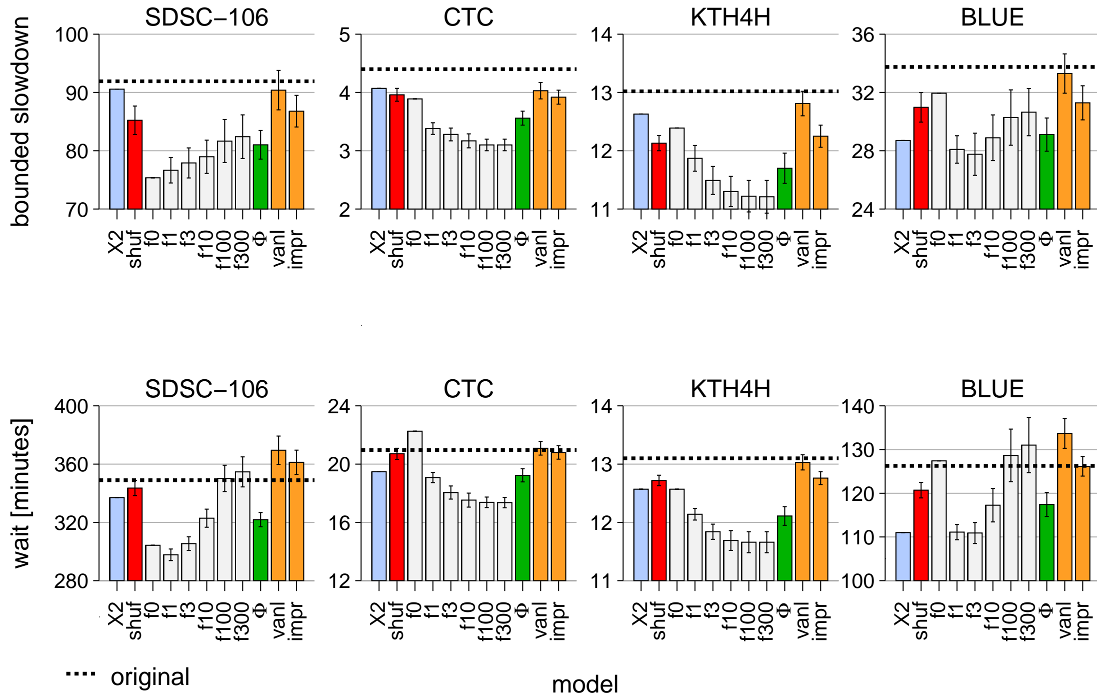


Figure 5.13: *Validating badness. The reason for the peculiar results associated with the average wait time of SDSC and BLUE remain unknown.*

- The f-model (see Sec.5.2). In accordance with [108], six values of f were chosen: 0 (complete accuracy), 1, 3, 10, 100, and 300.
- The Φ-model (see Sec.5.2).
- The *vanl*-model: the vanilla setting of the model developed in this chapter (defined above).
- The *impr*-model: the improved setting of our model, supplying it with some additional information (defined above).

Notice *X2* and *shfl* aren't models per-se as both are based on real estimates. The "competitors" of our model are f and Φ (producing estimates based on runtime).

Performance results are shown in Fig. 5.13 in the usual form of average wait time and bounded slowdown. The black dotted lines present the results of running the simulations using the original data. Therefore, models that are closer to this line are more realistic. Recall that our aim here is not to improve performance. Rather, it is to produce trustworthy results that are closest to reality. All the results associated with models that contain a random component (all but *X2* and *f0*) are the average of one hundred different simulation runs employing different seeds. The error-bars associated with these models display the absolute-deviation (average of absolute value of deviation from the average).

When examining Fig. 5.13, it is clear the two variants of our algorithm are more realistic, in that they are closer to the real thing (compare with f-s and Φ). Another observation, which reaffirms our results from Chapter 3 is that using increased f-s to model increased user inaccuracy

(for the sake of realism) is erroneous, as *f0* usually produces results that are much closer to the truth. In fact, *f0* is usually comparable to the results obtained by our model with the exception of the SDSC trace. However, this is limited to the FCFS-based EASY scenario: if introducing a certain amount of limited SJF-ness to the scheduler (e.g. as in SJF, SJBF, LXF&W, etc.), *f0* yields considerably better performance results in comparison to the original, whereas our model stays relatively the same (not shown). Another scenario in which *f0* can't be used is when evaluating system-generated runtime predictors that make use of estimates to make predictions (as in Chapter 4). Finally (returning to the context of EASY), unlike *f0*, our model has room for improvement as will shortly be discussed, and we believe it has potential to "go the extra mile".

A key point in understanding the performance results is noticing that the vanilla setting of our algorithm is surprisingly more successful in being closer to the original than its improved counterpart. This is troublesome as our entire case is built on the argument that models that are more accurate would yield results that are closer to the truth. The answer to the riddle is revealed when examining the *shfl* model. The fact of the matter is that one cannot get more accurate than *shfl*, as it "generates" a distribution that is *identical* to that of the original. Yet it too seems to be inferior to our vanilla model. This exposes our independence assumption (the random shuffle algorithm) as the true guilty party which is responsible for the difference between *impr* and the original. The correct comparison between *impr* and *vanl* should actually be based on which is closer to *shfl*, not to the original, as only with *shfl* can independence be assumed. Based on this criterion, *impr* is consistently better than *vanl*.

Once this is understood, we can also explain why the performance of *impr* (in terms of wait and slowdown) is always better than that of *vanl*. Consider the difference between the two models: *impr* simply has much more accurate data regarding *shorter* jobs (e.g. KTH4H's 25% of 5 minutes jobs). As short jobs benefit the most from the backfilling optimization, *impr* consistently outperforms *vanl* (in absolute terms).

5.12.4 Repetitiveness is Missing

We are currently not interested in artificially producing worse results by means of "falsely" boosting up estimates (as is done by *vanl* with respect to *impr*). This would be equivalent to, say, increasing the fraction of jobs that estimate to run E_{max}, which can arbitrarily worsen results. Our current goal is creating a reliable model. The above indicates that the problem lies in the assumption of independence, namely, the manner we assign estimates to jobs. While it is possible that this is partially because we neglected to enforce the accuracy to be as displayed in Fig. 1.4 (the accuracy histograms of even *shfl* are dissimilar to that of the original), we conjecture that the independence assumption is more acute.

It has been known for over a decade that the work generated by users is highly repetitive [48] (a fact which was exploited in the Chapter 4 to make predictions). Recent work [173, 133, 148] suggests that the correct way to model a workload is by viewing it as a sequence of *user sessions*, that is, bursts of very similar jobs by the same user. This doctrine suggests that a correct model cannot just draw values from a given distribution while disregarding previous values as is done by most existing parallel workload models (e.g. [30, 77, 21, 99]). The rationale of this claim is that the repetitive nature of the sequence within the session may have a decisive effect on performance results (as in the example we have given in Sec. 1.3.4, where a change of 30 seconds in one job resulted in a 8% change in the average performance of *all* due to the flurry session).

Since users tend to submit bursts of jobs having the same estimate value (Fig. 1.6, page 14), the end result is somewhat similar to that of the existence of estimates modes, but in a more "temporal sense": At any time instance, jobs within the wait-queue tend to look the same to the scheduler, as jobs belonging to the same session usually share the same estimate value. Consequently, the scheduler has less flexibility in making backfilling decision and the performance is negatively effected. Our *shfl* algorithm, along with all the rest of the models, do not entail the concept of sessions and therefore result in superior performance in comparison to the original.

To make out model complete, one must first develop a session-based model. This work is underway, but is far beyond the scope of this dissertation [148, 130].

5.13 Conclusions

For the conclusion of this chapter, we refer the reader to Section 7.2 (page 124).

Chapter 6

Workload Flurries and Data Sanitization

6.1 Introduction

Context The performance of a computer systems is a product of the workload to which they are subjected, as much as it is a product of their design and implementation [40]. Indeed, different workloads may lead to different absolute performance numbers, and in some cases to different relative ranking of systems or designs. Using representative workloads is therefore crucial in order to obtain reliable performance evaluation results. One canonical way to obtain representative workloads is to use logs that record the activity experienced by real productions systems. These can then be used as follows

- If a recorded system has a similar functionality to a new system being evaluated, one can assume that the same workload may apply. One can therefore "play back" the recording to drive a simulation of the new system and use the results as predictors of performance.
- Even if the new system differs from existing systems, the recorded workloads can be valuable: if the new system design is shown to produce good results when applied to a wide range of recorded workloads, one has strong indication that the results are general and representative. (This is largely the approach we have taken in this dissertation.)
- Alternatively, recorded workloads can be used as the basis for constructing a workload model, as was done in the previous chapter. This has two benefits. First, it often reveals insights and understandings that may lead to a better system design. Additionally, the output of a model can be put to use within the above two items; in this context, a model allows a more flexible usage than an actual log, as its parameters can be easily varied to reflect different system configurations.

All of the above are standard, heavily used, methodologies. Researchers share many logs of a wide variety of computer systems and use them to improve those systems in the aforementioned manner. The logs are routinely used as-is, no questions asked. The fundamental justification for this approach is the perception that recorded workloads reflect actual events that really did happen. And if it happened, it is "representative" (of the type of events the associated system must handle), and therefore must be included in the evaluation. We challenge this perception, and the remainder of this section is devoted to explaining why.

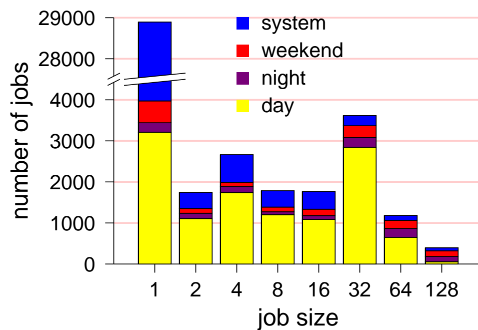


Figure 6.1: *Histogram of job sizes from the NASA Ames iPSC/860, showing abnormally many single-node system jobs.*

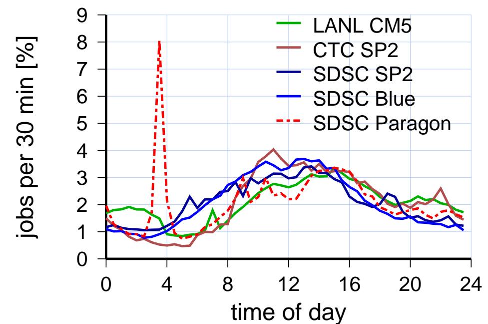


Figure 6.2: *Daily arrival pattern on 5 parallel supercomputers, showing abnormal spike at 3:30 AM on the SDSC Paragon.*

Bogus System Activity Large-scale systems often require continuous support from the vendors that installed them. In some cases, a vendor employee is even stationed at the installation site, so as to be on hand in case of need. Such employees also perform monitoring tasks and take preventative measures to avert failures before they happen.

Given the presence of such system staff, the workload observed on the system is actually a mixture of two classes of workload: work submitted by real users (the system's "payload"), and work submitted by the system personnel as part of performing their tasks. What we do about this depends on our goals. If we are only interested in user activity, system staff activity should be filtered out. But if we are interested in the complete system, then monitoring and maintenance tasks should be left in, because they are indeed part of what the system has to do.

However, sometimes system staff generate extraneous workload that is obviously bogus. One striking example was reported in the analysis of the workload on the NASA Ames iPSC/860 hypercube [48]. The histogram of job sizes on that 128-node machine indicated that more than half of the jobs were serial; moreover, most of the serial jobs were flagged as being run by the system support staff (Fig. 6.1). This turned out to be a result of an ad-hoc method used to verify that the system was operational and responsive by running the Unix `pwd` command on a single node. Overall, a full 56.8% of the trace (24025 jobs) were such check-runs. This type of activity was not observed on any other parallel system. It is site specific. Thus, it is quite obvious that these jobs should be removed if the trace is used to analyze or evaluate parallel workloads in general.

Non-Representative Robot Activity Another example is shown in Fig. 6.2. This compares the daily arrival cycle on 5 different parallel supercomputers. All display the expected periodic behavior, with load peaking during work hours and lower loads at night. But the SDSC Paragon machine has an additional and much higher peak between 3:30 and 4:00 AM. Upon investigation, it turned out that a set of 16 jobs with a distinct profile was executed during this time slice every day. While specific information is not available, it is reasonable to assume that these jobs served some system administration function and were executed automatically. It is again obvious that they should be removed when using the log for evaluations, so as to reduce the danger of optimizing for this abnormal behavior.

Flurries We believe the above are "easy to digest" examples, in that most analysts would agree that the associated abnormal data should be sanitized before being used. This chapter is largely devoted to a previously unknown, less obvious anomaly, called "workload flurries". Flurries consist

of rare, huge surges of repetitive activity by single users that dominate the workload for a limited time (see more precise definition in Section 6.3). They have two types of effects on performance evaluation. One is in the context of workload modeling, and specifically the fitting of statistical distributions to workload data. The existence of a flurry may alter workload statistics, leading to the use of un-representative values by an unwary analyst. The other is an effect on performance evaluation results when using the workload trace to drive a simulation. Flurries may cause a simulation to be very sensitive to fine details of the system configuration or workload, because the whole flurry reacts to a change *en masse* and thereby amplifies its effect. Hence extremely small modifications may lead to large effects that are not reliable predictors of real performance.

Roadmap We start in Sec. 6.2 with a detailed example concerning a real workload trace that spans two years and records 73,496 parallel jobs. We show that shortening the runtime of a *single* 18-hour job by a mere 30 seconds results in an 8% change in the average slowdown of *all* the jobs, solely due to the effect it had on a subsequent 375-job flurry that was submitted by a single user over a period of 10 hours. This motivates the study of flurries as unique and important events in computer workloads in Sec. 6.3. In Sec. 6.4 we show that “cleaning” or “sanitizing” workloads by deleting the flurries leads to more stable, reliable, and consistent performance evaluation results. In Sec. 6.5 we show why the removal of flurries is methodologically sound (namely, the correct thing to do). Flurries also have a detrimental negative effect on modeling activity, as shown in Sec. 6.6, further motivating their removal. Lastly, in Sec. 6.7, we show that the flurries phenomenon is not unique to parallel supercomputers and that it is in fact widespread.

6.2 A Case Study of Instability

In this section we present a case study showing how the presence of a flurry leads to unstable results: very small changes to the workload are amplified by the flurry and lead to an unexpectedly large change in the results. This example uses the SDSC-SP2 log.

6.2.1 Example of a Butterfly Effect

The largest user runtime estimate (E_{max}) appearing in the SDSC-SP2 trace is 18 hours, a limit imposed by the site administrators. Consequently, as jobs are killed once their estimate is reached, the longest jobs in SDSC-SP2 are limited to 18 hours. However, as we have shown in Fig. 4.2 (page 63), in a real system, it takes some time to propagate the instruction to kill a job to all the nodes. Therefore the trace indicates that some jobs run for a bit more than 18 hours. Of the 73,496 jobs in the trace, only 619 (less than 1%) have runtimes longer than 18 hours.

In a simulation it is possible to change the irregular runtimes to be *exactly* 18 hours. Surprisingly, we found that the average bounded slowdown is rather sensitive to such a change. The following is a striking example that demonstrates this phenomenon. The attributes of job 64,241 are listed in the left of Fig. 6.3. In our simulation, we have truncated this job’s runtime by a mere 30 seconds, and set it to be exactly 18 hours (a modification of 0.0463%). This was the *only* change we’ve made, that is, we have modified one job out of 73,496 (0.00136%). Remarkably, as a result, the average bounded slowdown of *all the jobs in the trace* changed from 88.16 to 81.38 — that is, by about 8%! Moreover, the effect turned out to be dependent on exactly how much the runtime of this job was changed. Fig. 6.3 shows the effect of different changes to the runtime of job 64,241

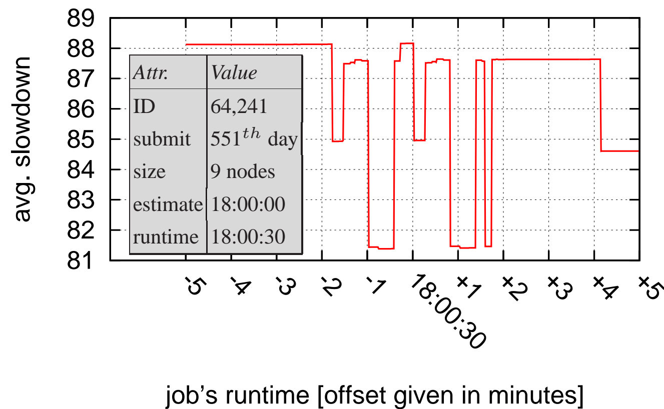


Figure 6.3: *Average bounded slowdown (obtained by simulating EASY on the SDSC-SP2 trace) as a function of the simulated runtime of the specified job. The job's original runtime is 18:00:30 and so an offset of +1 means the simulated runtime is 18:01:30.*

on the overall average. Note that counterintuitively, the average may change by roughly the same amount both by enlarging and by reducing the runtime of the job.

6.2.2 The Role of a Flurry in Causing the Effect

In a nutshell, the mechanism leading to the above effect has two components. First, the backfilling algorithm propagates the small modification to a single job and influences many other jobs. Second, a whole flurry of similar jobs are affected *en masse*, and their combined weight leads to the observed change in the global average.

In a batch system, a reduction of 30 seconds in the runtime of a job has the obvious immediate (minor) effect of allowing other waiting jobs to obtain the required resources sooner, possibly allowing them to start earlier by up to 30 seconds. But in the context of a backfill scheduler, a more important effect is that a modification of 30 seconds is enough to make the difference regarding a backfilling decision: by terminating 30 seconds earlier, a slightly larger window is opened, and a job that was previously considered too long to be backfilled may now fit into the available space. This causes a modification of the schedule down the road. Such a chain of modifications allows the effect of one truncated job to accumulate. In our simulation, exactly 2024 jobs were affected by the truncation of the runtime of job 64,241 (in terms of changed start time). The changes in start time are depicted in Fig. 6.4, where each affected job is represented by a single point. The rest of the schedule remained unchanged.

According to the figure, many of the affected jobs have almost zero difference in start time, and probably reflect the fact that 9 processors became available 30 seconds earlier. The bigger differences between the original and modified schedules are focused in two areas: between days 560-570 (10 days after job 64,241), and between days 580-585 (a month after), and reflect changes in backfilling decisions. Nevertheless, the 8% change in the average bounded slowdown actually stems from start-time differences associated with a group of jobs submitted on the 581st day. This can be seen in Fig. 6.5, that compares between the running averages of the bounded slowdowns obtained by the two schedules. (The running average at time T is defined to be the average of

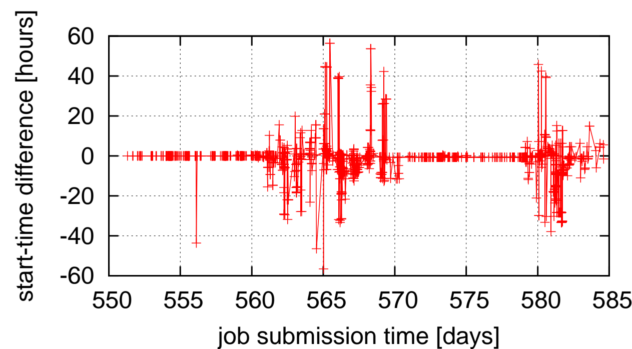


Figure 6.4: *Shortening job 64,241 by 30 seconds had an effect for more than a simulated month, causing 2,024 subsequent jobs to start earlier or later by up to nearly 60 hours. (Only these jobs are shown.) The Y-axis indicates the difference between the start time of jobs in the original and truncated simulations (negative values indicate jobs that started earlier due to the truncation).*

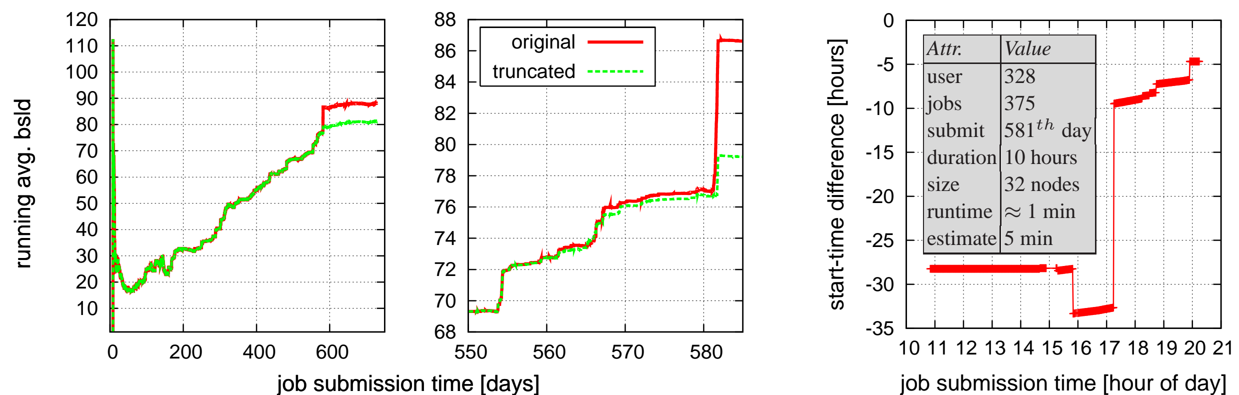


Figure 6.5: *Running average of the bounded slowdown obtained by the EASY scheduler on the SDSC SP2 trace with/without the 30-second truncation of job 64,241. Left: full trace. Right: zoom in on the part where start-time differences occur.*

Figure 6.6: *Start-time differences of the specified flurry jobs by user 328 (X-axis denotes hours on that day).*

bounded slowdowns experienced by jobs that were submitted prior to T.) From this figure it is evident that the major difference in overall average performance was due to changes associated with jobs that were submitted at the 581th day, and that all the other changes (e.g. between days 560–570) had a negligible effect.

A closer inspection of the data revels that the perceived change is due to a flurry composed of 375 similar jobs that were submitted sequentially over a period of about 10 hours in the 581st day (exactly one month after the truncated job was submitted). All these jobs were submitted by user 328, required 32 nodes, were estimated to run five minutes, and ran for about one to two minutes; this is the biggest flurry shown in the right of Fig. 6.9. The running average of the bounded slowdown of the original and the "truncated" runs were quite similar when the first job of this flurry was submitted (about 1% difference). By the time the last job of the flurry was submitted, the difference was as high as 9%.

Fig. 6.6 shows the start-time differences associated with the jobs of the flurry (this is a subset of the data displayed in Fig. 6.4). The jobs' profile similarity along with the fact that they were

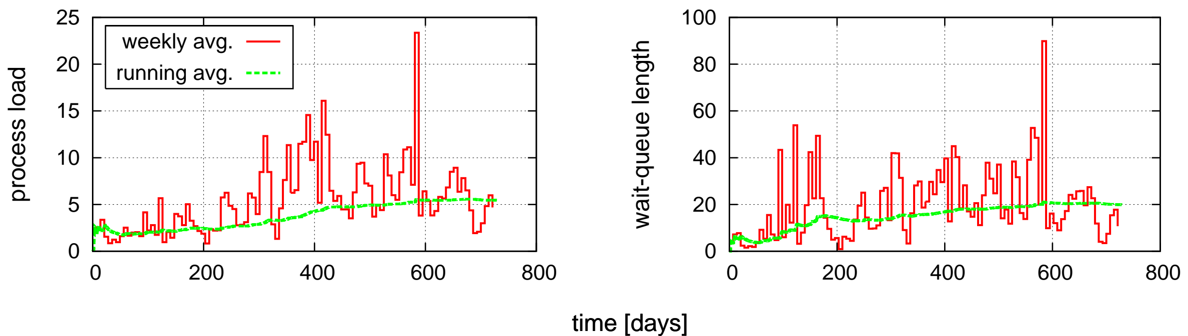


Figure 6.7: *Evolution of the SDSC SP2 process load and the waiting-queue length when simulating EASY on the original trace.*

submitted sequentially, explains their tendency to be effected in the same way by changes to the schedule (in terms of wait time). Note that the effect of shortening the wait time of a job with runtime of around one minute by 30 hours is a reduction of 1800 in its bounded slowdown. This is a huge figure compared to the average bounded slowdown of the entire trace (less than 90), a fact that explains the considerable difference between the slowdown averages of the truncated and original runs.

6.2.3 Explaining the Sensitivity

Truncating the runtime of job 64,241 (which was submitted 30 days before the flurry) is only one of many trivial modifications we have identified that resulted in a significant change in the average bounded slowdown. These modifications may involve more than one job, and may be applied to jobs with different runtimes, different runtime estimates, and different sizes. However, all these modifications have an effect only when the flurry identified above is scheduled. No other flurry in this log displayed this type of sensitivity. In particular, similar modifications in the neighborhood of the huge flurry identified at the beginning of the log (see Fig. 6.8) didn't produce similar effects, even though this flurry is an order of magnitude bigger than the flurry above (in terms of the number of jobs composing it).

The reason that the 375-job flurry is so sensitive is that it induces a very high process-load on the system. The process load at time T is defined to be the total number of running or waiting processes (not jobs) that are present in the system at that time instant, divided by the size of the machine. For example, if a machine with 10 nodes is currently running 8 processes (leaving 2 nodes idle), while two jobs of size 6 are waiting in the queue, then its process load is $(8+6+6)/10 = 2$. The left of Fig. 6.7 displays the evolution of the process load associated with the SDSC SP2 trace. The unequivocal peak in the weekly-average line occurs in the week that contains the 581st day. The right of Fig. 6.7 shows that this is also reflected in the state of the waiting-queue.

We note in passing that the long-term average process load grows continuously across the trace. This explains the growth trend of the average bounded slowdown (as seen in the left of Fig. 6.5). It is tolerated by the users because the majority of the jobs still enjoy a fairly reasonable quality of service, as indicated by the bounded slowdown median of the SDSC SP2 trace which is 1.8.

Finally, it should be noted that the effect described above depends on the existence of the flurry, but not only on it. It also depends on the metric being used. When measuring the actual response time, for example, the difference caused by the flurry jobs is not significant enough to change the

overall average, because the average response time is dominated by long jobs [40]. By contrast, the average slowdown is dominated by short jobs (that typically have higher slowdowns), so a flurry of short jobs may have a large effect.

6.3 The Phenomenon of Workload Flurries

Having seen the effect that workload flurries may have on performance evaluations, we now turn to the phenomenon of workload flurries themselves. We define a workload flurry to be a pattern of activity with the following characteristics:

1. it causes a level of activity significantly higher than usual, thus dominating the workload,
2. it exists for a limited period of time,
3. it significantly changes the distributions of workload attributes, and
4. it is caused by a single user.

The name "workload flurry" derives from the first and second attributes, and from the fact that the items constituting the flurry are typically lightweight, because otherwise the system would be overwhelmed by their numbers. The above definition is derived from observations of such phenomena in the long-term workloads experienced by large-scale production parallel supercomputers, as demonstrated now. However, we believe that the phenomenon of workload flurries is widespread, and indeed we have also found such flurries in other system types (see below).

Fig. 6.8 shows the job arrival rate at the granularity of weeks of 6 logs (see details in Chapter 2). In all of them, large flurries are observed. They range in size from double the average activity to 10 times the average activity, are caused by a single user, and extend from a few days to several weeks. The flurries in the CTC log and the Blue Horizon log seem similar to normal fluctuations, but nevertheless turn out to have an important effect (at least for CTC), as shown in Section 6.4. It should be noted that flurries were observed in all the long logs in the Parallel Workloads Archive, but were not observed in the shorter ones. Indeed, periods several months long with no flurry occur also in the logs that do include flurries.

Fig. 6.8 shows an especially prominent flurry in the SDSC-SP2 data. But this is *not* the flurry that caused the instability described in Section 6.2. Rather, that flurry is a process flurry, i.e. it includes very many processes but not so many jobs. Fig. 6.9 illustrates the weekly process arrival rate on two of the machines, showing that process flurries do not necessarily correspond to job flurries (the largest one in SDSC-SP2 corresponds to the flurry that caused the butterfly effect, above). In fact, what exactly constitutes a flurry depends on the context in which the question is asked. The "high level of activity" (mentioned as part of the definition of flurries) can in principle also be defined in terms of memory usage, disk operations, or network bandwidth consumed.

The statistical nature of the observed flurries is explored in Fig. 6.10 (representative for other logs as well). This shows the joint distribution of two major attributes of parallel jobs: the number of processors they use, and their runtime. The flurries tend to correspond to specific locations in these scatter plots, indicating that they are largely composed of jobs with fixed characteristics. In particular, the jobs composing the flurries identified here tend to be small, using few processors and/or running for a relatively short time, as witnessed by the fact that they concentrate near the axes (note that both axes use a logarithmic scale). The fact flurry jobs' attributes are distinct from that of the other jobs, has a profound affect on modeling, as will be discussed below.

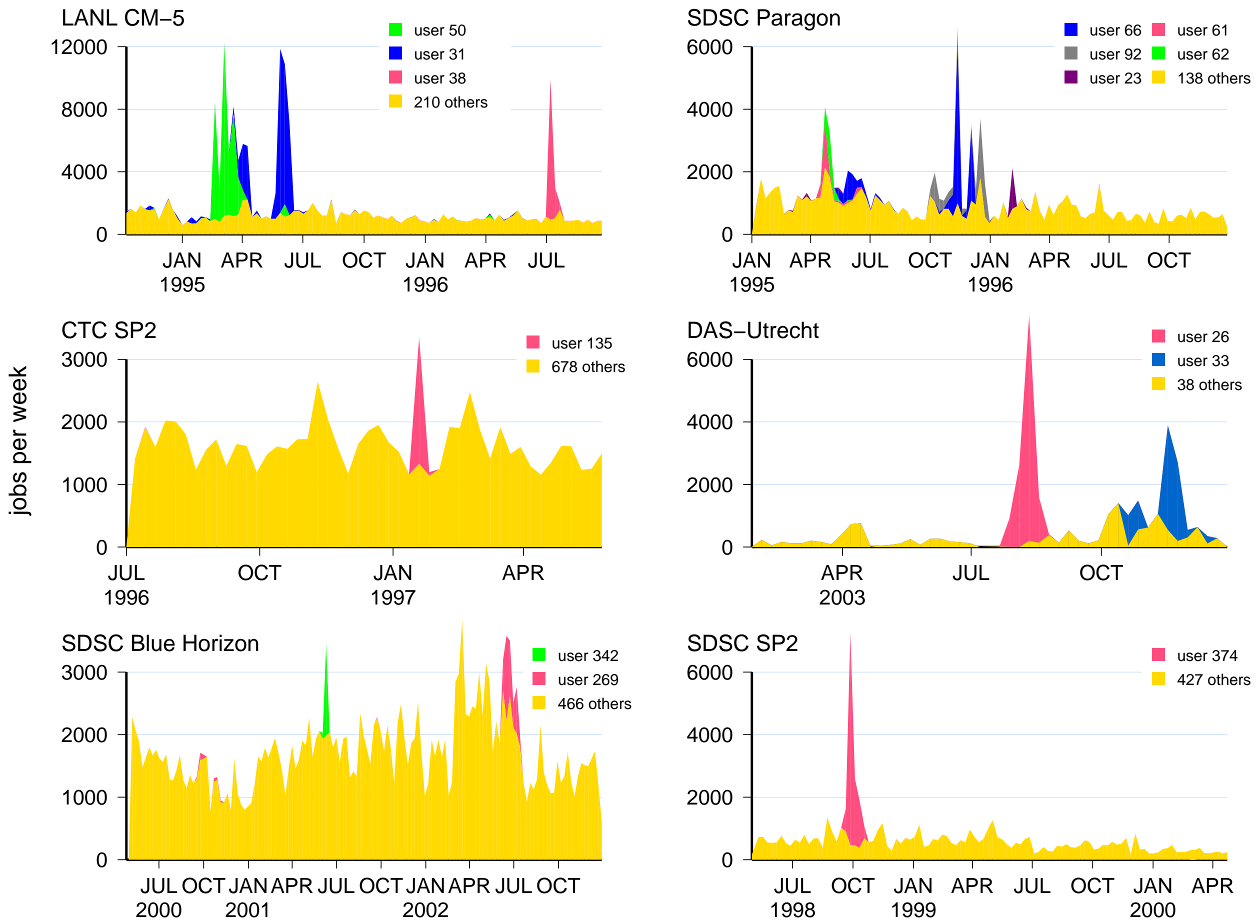


Figure 6.8: *Per-week job arrivals in six parallel machines. All exhibit flurries of activity due to single users.*

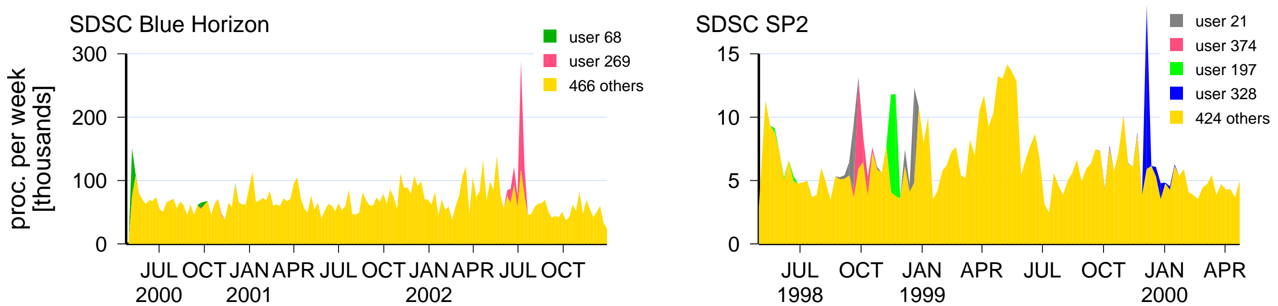


Figure 6.9: *Process arrivals per week also display flurries (can be different from the job-arrival flurries).*

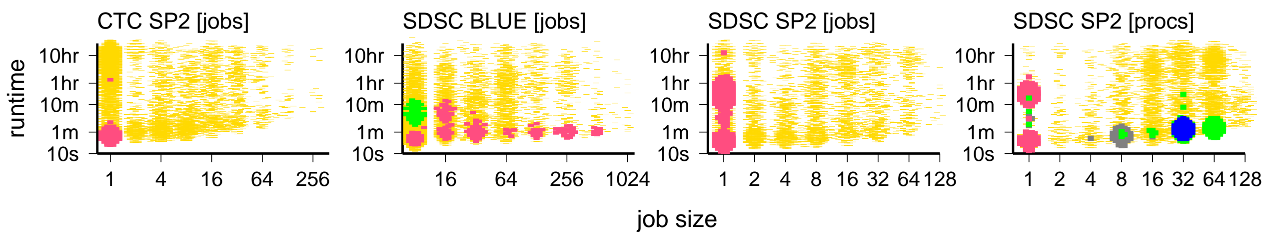


Figure 6.10: *Scatter plot showing joint distribution of job size and runtime. The single-user flurries typically have rather unique characteristics. Color-coding corresponds to flurries shown in Figs. 6.8 and 6.9.*

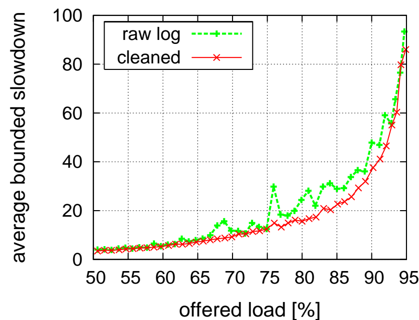


Figure 6.11: *Simulation of EASY on CTC. Flurries tend to be sensitive to exact simulation conditions, leading to instabilities. Simulations using a cleaned log are smoother.*

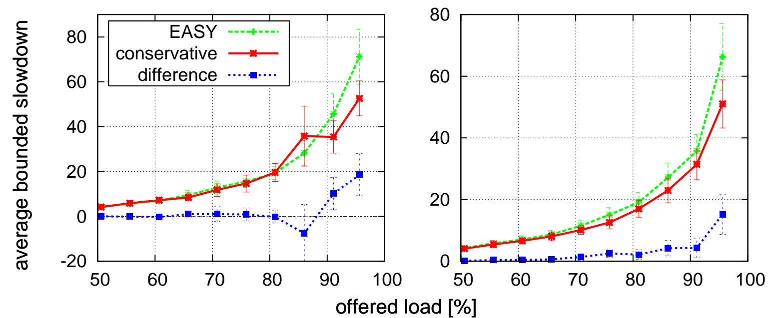


Figure 6.12: *Comparison of EASY and conservative backfilling, using the CTC log and accurate user runtime estimates. Left: using the complete log leads to inconsistent results. Right: removing the user 135 flurry leads to consistent results showing conservative backfilling is preferable for this scenario.*

6.4 Impact of Flurries on System Evaluation

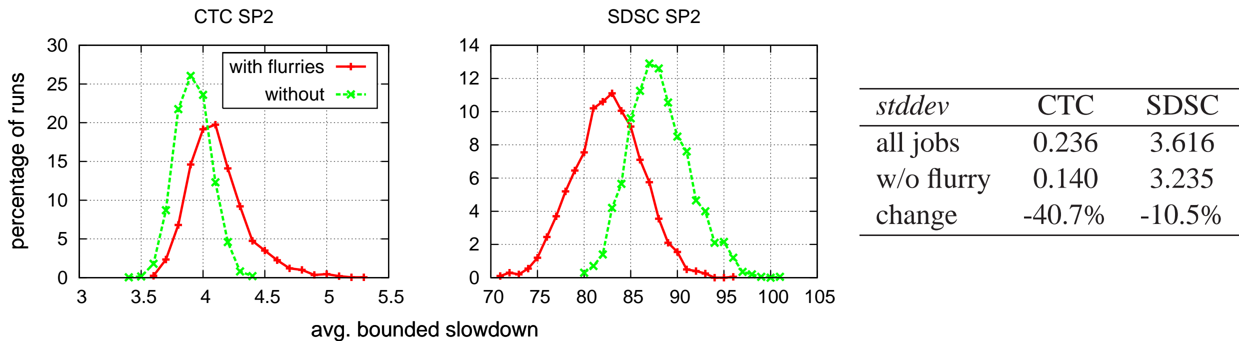
As we've seen, simulations of parallel job scheduling can be extremely sensitive to the exact workload conditions. This may also happen in normal evaluations, without any targeted modifications such as the truncation of job 64,241 as described above.

An example is given in Fig. 6.11, using CTC. This is again a simulation of the performance of EASY backfilling, this time showing how it depends on the system's offered load (varied as described in Chapter 2). As Fig. 6.11 shows, changing the load causes large fluctuations in the bounded slowdown results when using the raw log. It would be ludicrous to take such effects at face value, and claim that, say, the expected performance at a load of 77% is much better than at a load of 76%. In fact, these fluctuations are again examples of flurry amplifications: if the 2000-job flurry of activity by user 135 shown in Fig. 6.8 is removed (this is 2.5% of the total of 78,500 jobs in the log) the result becomes a smooth curve similar to those produced in queuing analysis.

Given that results such as these are hard to predict and correlate with the modifications used to change the offered load, they can also sway the results of evaluations. An example is given in Fig. 6.12. This shows a study comparing EASY backfilling with conservative backfilling (reservation to only the first, or all of the queued jobs, respectively; see Sec. 1.1.2 for details). The study in question dealt with the effect that the accuracy of user runtime estimates have on the performance of the two backfilling schemes [40]. The results shown in Fig. 6.12 (left) were obtained by simulating the CTC workload using accurate runtime estimates, rather than real user estimates. The results were inconsistent, showing that conservative backfilling produce higher slowdown values for an offered load of 85% but lower values for 90% and 95%. This inconsistency was traced to the same flurry identified above: rerunning the simulations on a modified workload where the flurry was removed led to the cleaner results shown on the right.

We note that the "difference" curve is not the difference between the performance *averages* obtained by EASY and the conservative algorithm, as this is still not statistically significant (notice that the associated 90% confidence intervals are still overlapping). Rather, it is the result of a more sophisticated analysis, using the common random numbers[1] variance reduction technique [89]. In

[1]The name is somewhat of a misnomer in this case, as we use a logged workload rather than generating it using a



stddev	CTC	SDSC
all jobs	0.236	3.616
w/o flurry	0.140	3.235
change	-40.7%	-10.5%

Figure 6.13: *With randomization, simulation results become non-deterministic. Flurries make them spread out more, reducing the accuracy with which results can be reported.*

this analysis, we first compute the difference in slowdowns between the two schedulers on a per-job basis (which is possible because we are using the same workload trace). We then compute confidence intervals on these differences. This shows that the flurry indeed makes a big difference in the quality of the results. When it is present, we cannot say anything definite for most offered loads, as the confidence intervals for the difference include 0. When it is removed, the advantage of conservative over EASY is clear across the whole range of offered loads.

A third example is given in Fig. 6.13. This is again part of the study of the effect of user runtime estimates, this time by randomly shuffling the estimates in the log among the jobs (for details, recall Sec. 5.12.2, page 99). Due to the shuffling, the average slowdown is different in each run. The figure shows the histogram of these averages over 2000 runs. When flurries are present, the standard deviation is larger, thereby enlarging the confidence intervals characterizing the result.

6.5 On Why the Removal of Flurries is the Right Thing to Do

A common initial reaction to the notion of referring to the activity of some particular user as non-representative and unreliable, is of disbelief. The underlying rationale of this view is that one can't get more reliable and representative than an activity which was actually recorded on a real system. This section addresses this concern in an incremental manner. It discusses the goals of the performance evaluation process and the meaning that the terms "representative" and "reliable" have within this context. It then shows why the removal of flurries inherently coincides with these goals and ideas.

A party that opposes the removal of flurries acknowledges the fact that the observed instability renders the results (of the corresponding performance analysis process) useless. Indeed, with respect to the overly-sensitive system under consideration, one cannot dispute the fact that a negligible perturbation can sway the results in the other direction, deeming them unreliable and useless. The opposing party therefore contends that the correct conclusion should simply be that "there are no valid conclusions". Alas, no-conclusion is a common and widespread result, as anybody that ever conducted system research knows too well. The question is whether this reasoning applies if the inconclusiveness of the results is due to flurries. We contend it does not.

random number generator.

6.5.1 How About Removing Entire Days?

A first "rough" reason why flurries should not be show-stoppers for the performance evaluation process is that they are temporally confined in relatively short periods of time and have short-lived impact. Consider, for example, the 24-months SDSC log, which is the log that exhibits the heaviest load conditions (see Table 2.1, page 32). In Sec. 6.2 we have shown that the instability and sensitivity is due to the 375-jobs flurry that was submitted during the 581th day. It turns out that even under SDSC's heavy load conditions, removing this flurry has a lasting effect of *only five days*, namely,

1. the schedule that is produced when using the original SDSC as the simulator's input, and
2. the schedule that is produced when deleting the aforementioned 375 jobs from SDSC and using this "cleaned" version as the simulator's input

are *identical* in every respect until the 580th day and from the 586th onwards. Thus, a perfectly legitimate methodology would be to delete days 581-585 entirely from the log and proclaim that the results of the evaluation only apply to 725 days out of the 730 that the log covers. This statement is sound, and it has real value: it is an accurate performance analysis that applies 99.3% of the the time. It should be obvious that this result is far superior to the "no-conclusions" alternative that the opposing party has to offer. Indeed, an important goal of the performance analysis process is e.g. to help choose between competing systems. A result that applies 99.3% of the time can be rather helpful in this respect, especially if the alternative is to not say anything at all.

Our justification for removing the 5 days is that they are (literally) not "representative" of the other 725. Since the latter are the vast majority, it is justified to characterizing them as "the norm". Likewise, it is justified to characterize the 5 days which are affected by the flurry as a rare short-lived "anomaly". Finally, the assertion that an evaluation which includes the anomalous 5 days is unreliable, is also well justified; in this sense flurries may indeed be characterized as an "unreliable" activity. Importantly, one flurry is not representative of another: all the flurries we have encountered are substantially different from one another (in terms of the number of jobs that compose them, the attributes of the jobs, and the duration in which they are submitted). We therefore contend that flurries are very similar in essence to the site-specific anomalies described in Sec. 6.1, which most analysts would agree to sanitize due to the fact that they are not representative of the typical workload.

6.5.2 Standard Alternatives are More Aggressive

Deleting a few short periods from the input is a actually a very subtle approach compared to several other routinely practiced (and much more aggressive) standard methodologies. For example, in computer architecture, in the interest of shortening simulation time, it is customary to choose a few dozens of relatively short "representative" instruction sequences of a SPEC application [143], "stitch" these sequences together, and use the result instead of the original application under the assumption that the former is reasonably representative of the latter [114, 117]. The short version might be less than 1% of the total. Choosing small fractions of the input to represent the entire input constitutes a far more aggressive filter than removing only one fraction and explicitly proclaiming that the result does not apply to the removed part. And so, unlike our methodology by which one can be sure that the obtained results are representative for most of the time, with the stitching

approach, one can only hope that the selection is representative. (Also, note that had we randomly chosen e.g. 1% of supercomputer log, the chances of choosing the 581th day were rather slim.)

A related acceptable approach from the supercomputing domain is to divide the input (usually years worth of activity) into relatively short disjoint consecutive sequences (usually individual months) and to report the results associated with some “representative” subset of these subsequences [15] or even with all of them [108, 149]. Using only a subset of the months is obviously more aggressive than our approach of just deleting a few days of activity. Using all the months translates to localizing the effect of the flurry within the month in which it occurred. But reporting the outcome associated with this month as a reliable result is methodologically erroneous, because it does in fact contain a flurry that might arbitrarily distort the result as shown above. Thus, the correct thing to do is single out the result associated with the month that contains the flurry as unreliable, which is equivalent to our removing-a-few-days approach, but has the drawback of deleting some extra “innocent” days instead of just the “contaminated” ones.

Yet another standard methodology (again for the sake of shorter simulations) is to use some small prefix of the input as representative of the input in its entirety. This is done both in computer architecture, as well as in supercomputing related research [81, 123] and is considered acceptable, even though its actually far less reliable than stitching: Focusing only on the beginning of the input runs the risk of missing important aspects, and is certainly a more aggressive approach than using all of the input except a few days.

Finally, many research efforts simply abandon the use of real workload traces altogether [169, 170, 57] and prefer to use workload models [77, 20, 99] (that are based on real traces), even though the models are overwhelmingly more well-behaved than the real thing [39]; for example, they generate stationary distributions (even though this is seldom the case; see Fig. 6.7 and [17, 149]), they lack self similarity (even though this property was shown to consistently exist in real logs [151]), they often make various unrealistic assumptions [39, 157, 133], and they are certainly not generating any flurries. Consequently, results obtained through using existing workload models to drive a simulation is far less reliable than using real logs, even if the latter is a slightly reduced version of the original. Put in another way, using the output of models as the input for a simulation is equivalent to applying sanitization to the real thing that is orders of magnitude more aggressive than just deleting a few days.

6.5.3 How About Removing Just the Anomalous Part of the Days?

Having established the fact that the removal of a some short period from a log (in our case the said period is at least an order of magnitude shorter than the log) is a valid and sound methodology, we go on to further refine this methodology and make it even more subtle and even less intrusive. Note that, with the exception of the flurry jobs, the five days in which the flurry resides consist of perfectly regular activity. There is nothing which is fundamentally different between the jobs that reside outside the five days and the non-flurry jobs that reside within them. Importantly, the flurry activity is completely independent of the non-flurry activity, as it is largely generated by independent, unrelated, users. Indeed, as the flurry jobs are generated by one user, it is reasonable to speculate that they could have been submitted during a different period, had the schedule of the submitting user been somewhat different.

Thus, deleting only the flurry jobs and leaving the rest of the five-days activity in, is a founded and well-justified approach. It is certainly as valid as deleting the 5 days in their entirety, and

is actually superior in two respects. Firstly, it preserves all the available information which is reliable and representative and therefore allows the performance analyst to argue for a result that is somewhat stronger than “99.3% of the time”. The second advantage of removing just the flurry jobs is that it avoids the issue of determining the exact duration of the period to be deleted: In the above example, we had to simulate the run with and without the flurry activity, and compare the results on a per-job basis in order to determine that the last affected job resides in the 585th day. Note that this period can possibly change under different circumstances, e.g. if we used a scheduler that is different from EASY, if we artificially changed the load, etc. Determining the exact duration can therefore be even more labor intensive. Consequently, deleting only the flurry jobs is a much simpler alternative (in fact it’s quite simple even in absolute terms) and therefore it has a viable chance to actually be adopted, especially if analysts are provided with a clean log to begin with and are spared all the details.

6.5.4 How About Not Removing Anything and Separate Averages Instead?

While be believe the above arguments are more than enough to justify the removal of flurries, there is in fact an even more subtle sanitization methodology we can apply, which yields identical results to the approach we advocate (of deleting flurries), thereby further strengthening it. It turns out that we actually don’t have to remove the flurry to eliminate the unwarranted sensitivity. Instead, we can simulate the input as is, as long as we exclude the flurry jobs form participating in the overall average performance metric (slowdown in our case). The practice of separating the job population to disjoint subcategories (wide vs. narrow, short vs. long, backfilled vs. non-backfilled etc.) and presenting the separate performance averages associated with each, is very popular and heavily used [125, 115, 141]. Accordingly, we suggest that the subcategories should be flurry vs. non-flurry jobs. With this approach we do not alter the input sequence *at all*. Specifically, the flurry jobs are simulated along side the rest of the jobs and are allowed to influence them. The only change we introduce is in the manner by which the performance metric is computed: we isolate the performance experienced by flurry jobs in an average slowdown of their own.

Fig. 6.14 illustrates the result of doing this with the average slowdown obtained when repeatedly simulating SDSC, such that in each simulation the runtime of job 64,241 is slightly altered. This is exactly the experiment that was described in relation to Fig. 6.3, only now we add two curves that are associated with the average slowdown of the flurry and non-flurry jobs, respectively. As can be seen, the non-flurry slowdown average is stable between 79-80, whereas the flurry average “goes wild” between 450-1450. Indeed, the sensitivity as manifested in the highs and lows of the “all jobs” curve that is associated with the combined average of the two, is perfectly correlated with the highs and lows of the flurry curve. This again highlights the sensitivity effect as the product of hundreds of (flurry) jobs that react in a similar manner to a minor change and therefore disproportionally sway the overall average in their direction.

Fig. 6.15 compares the result obtained when systematically varying the load of two versions of the CTC workload: the raw log and a cleaned version of it (that deletes flurries). Like with the example given in the previous paragraph, the current example corresponds to an experiment we conducted earlier in this paper, the result of which was presented in Fig 6.11. Once again, the difference between Fig. 6.11 and Fig. 6.15 is that the latter adds two curves that subdivide the overall slowdown associated with the raw (non-sanitized) log into two: the average slowdown of flurry and non-flurry jobs, respectively. (The flurry curve is plotted in the right subfigure only, due

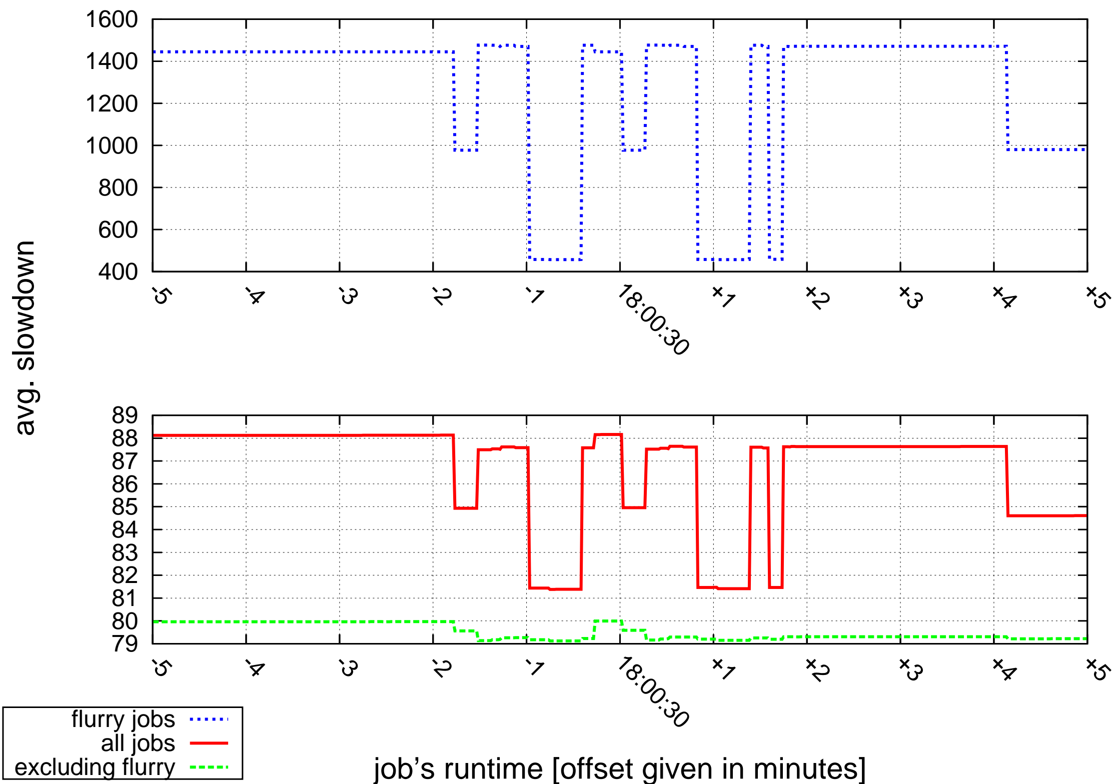


Figure 6.14: *Separating the overall average slowdown to flurry vs. no-flurry jobs, reveals that the later is quite stable and highlights the former as solely responsible for the instability. (Associated with Fig. 6.3.)*

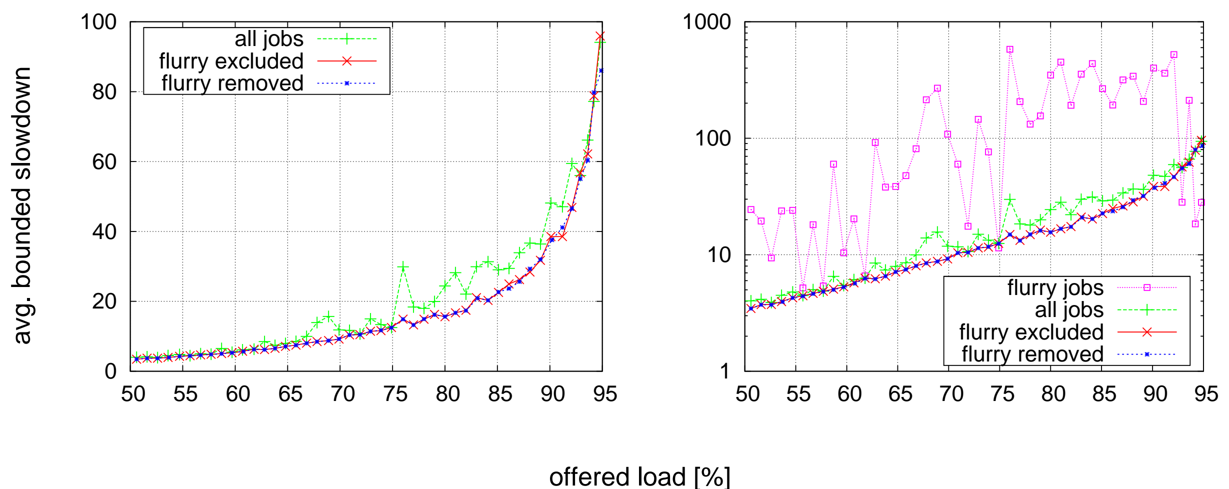


Figure 6.15: *Averaging only non-flurry jobs ("flurry excluded") has an identical effect to removing the flurry altogether ("flurry removed"), implying the latter approach is preferable due to its simplicity. The perfectly correlated highs and lows of the combined curve ("all jobs") and the flurry curve ("flurry jobs") single out the latter as the exclusive reason for the observed instability. (Associated with Fig. 6.11.)*

to the limited Y axis scale in the left.) Concentrating on the left subfigure, we see that there is virtually no difference between the average associated with the cleaned log ("flurries removed") and the one associated with the non-flurry jobs within the raw log ("flurries excluded"), as the two curves continuously overlap. Turning to the right subfigure, which introduces a widened log-scaled Y axis and can therefore add the flurry curve into the picture ("flurry jobs"), we again see that the latter's highs and lows correlate with those of the combined curve ("all jobs") and single out the flurry as the sole cause of instability.

The bottom line is that deleting the flurry jobs has an almost identical effect to leaving them in while isolating them within an average of their own (the least aggressive methodology). This result further justifies the flurry-deletion approach, which is preferable over the average-separation approach, as it is much simpler.

6.5.5 How About Not Separating the Averages and Shake the Input Instead?

A final alternative to dealing with the instability that is generated by flurries is what we call *input shaking*. With this approach we (1) leave the flurries in, and (2) do *not* separate the averages. Instead, we substitute a single simulation run with multiple runs, such that for each run we systematically introduce negligible random perturbations into the input, e.g. by changing the arrival time of 10% of the jobs to be $\pm u$ seconds earlier or later, where u is e.g. uniformly distributed between 0 to 60. We argue that a slightly perturbed version of a log is as representative and reliable as the original version. The multiple simulations create a sample space, which can be averaged and bounded within confidence intervals. In [161] we show that this approach is effective and largely defeats the instability generated by flurries. But a full description is beyond the scope of this dissertation. Shaking has the clear advantage of not requiring that flurries would be known. It allows the use of raw data as is, at the price of deciding upon the shaking specifics and of performing multiple simulations instead of one.

6.6 Impact of Flurries on Modeling

The above sections focused on the instability induced by flurries on the process of parallel systems evaluation. This section focuses on flurries' impact on modeling. As noted above, the fact flurries have statistical properties that are different from the "normal" background distributions (Fig. 6.10) has significant implication on this subject. Three examples are given.

Weekly Cycle The first example that demonstrates this is given in Fig. 6.16, which shows the weekly cycle of all 16 logs available through the Parallel Workload Archive [110], as listed in Chapter 2. Naturally, more work is being done on weekdays than on weekends, with the single exception of DAS-Utrecht. In fact, this is also true for the latter log, with the single exception of 4,297 jobs submitted by user 26 on Saturday Aug 16. (Which actually constitutes the major part of the associated DAS-Utrecht flurry shown in Fig. 6.8.) Indeed, when deleting this flurry, the weekly cycle of DAS-Utrecht becomes similar to that of all the other logs. Obviously, it is erroneous to base a workload model that takes into account the weekly cycle on the raw data of DAS-Utrecht, as throughout all the logs in all Saturdays, but one, the load is low.

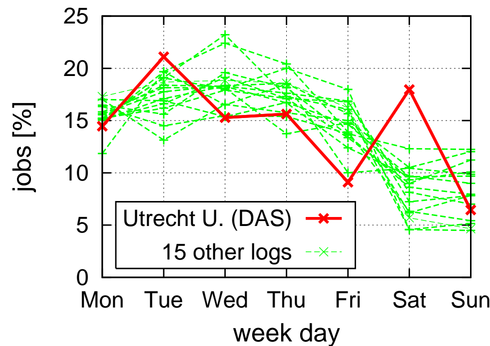


Figure 6.16: *Weekly arrival pattern on 16 parallel computers, showing abnormal spike Saturday within the DAS-Utrecht log.*

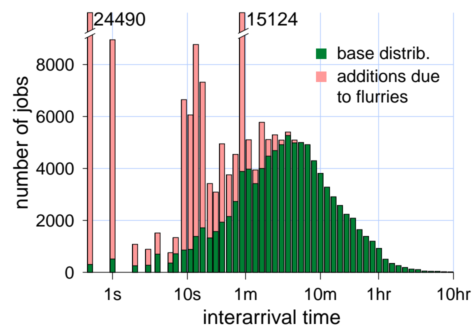


Figure 6.17: *Flurries turn the lognormal distribution of interarrival times in the LANL CM-5 log into a noisy and modal distribution.*

Interarrival Times Due to their repetitive nature, flurries tend to modify the workload distributions by adding huge modes. Focusing on the LANL CM-5 interarrival times as an example, we find that the distribution for the whole log is distinctly modal, with several values that are extremely common and each come from a different flurry (Fig. 6.17). After the flurry-related data is removed, the underlying distribution can easily be characterized as lognormal.

Non-Stationarity Flurries are not only different from the normal workload, but also different from each other. This combination leads to severe non-stationarity, as demonstrated in Fig. 6.18. The figure compares the distributions of four different workload attributes in the 1995 and 1996 portions of the LANL CM-5 log. For example, in 1996 the log contained a large flurry of activity by user 38 as seen in Fig. 6.8. The flurry consisted of jobs that were about 10 seconds long, arrived about 12 seconds apart, ran on 128 nodes, and used either very little memory or about 1.84 MB per node. This accounted for 12,344 (29%) of the total of 42,702 jobs in this part of the log, and thus had a decisive effect on the distributions of these workload attributes.

For comparison, during 1995 the log contained two other flurries, by users 31 and 50, which accounted for 71,161 (58%) of that year's total of 123,058 jobs. By comparing the 1995 and 1996 distributions in Fig. 6.18, we see that the workload seems to be non-stationary, as the distributions for the two years are quite different (dashed lines). But if the flurries are removed, we find that in reality the base workloads are actually quite similar to each other (solid lines). Thus the major differences between 1995 and 1996 are actually the result of flurries introduced by 3 users out of a total population of 213. Including the flurry data gives the actions of these 3 users significant sway over the results.

6.7 Generalizing

All of our examples so far come from the supercomputing domain. However, flurries are not unique to parallel supercomputers. Once we became aware of the phenomenon and began to look for it, it was rather easy to find it in other systems. Here three examples are provided, based on logs generated by three different types of departmental servers.

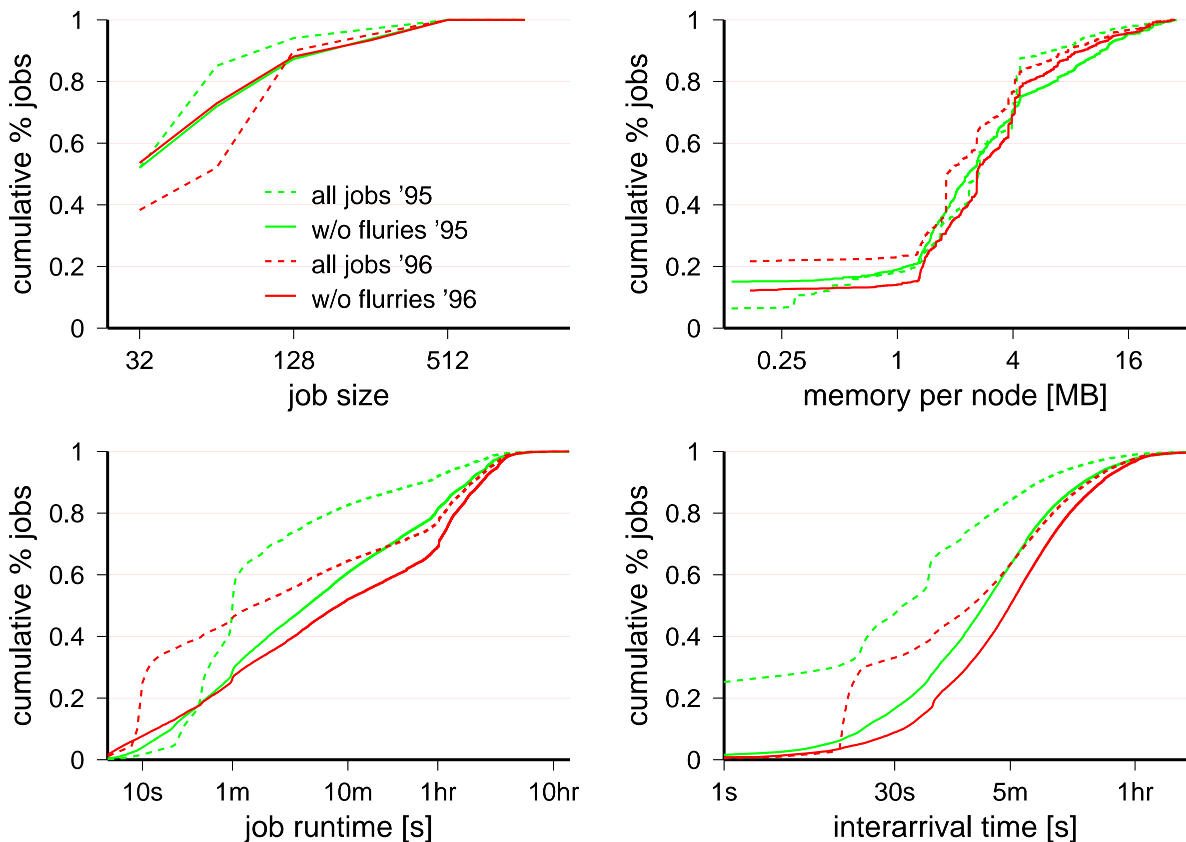


Figure 6.18: *Changes to the distributions of workload attributes when flurries are removed from the LANL CM-5 log. Most differences between 1995 and 1996 are attributed to the inclusion of flurries.*

CPU Server A large flurry was observed in the session log for March 2004 of a Unix server used by students (Fig. 6.19). This turned out to be the result of ftp'ing a large directory structure by a certain student one afternoon; the (MS Windows) implementation automatically opened a new ftp session for each directory, and this was logged as a distinct user session. Obviously, this data does not represent normal user sessions, and would cause misleading results if used as the basis of an attempt to optimize for interactive user sessions.

Authentication Server Another example is the activity on our departmental authentication server (Fig. 6.20). In this case data covering a long period was available, and two distinct flurries were observed. These were traced to a bug in Windows, where an authentication failure led to an infinite loop of retries. Indeed, it is possible that some of the flurries on supercomputers are also the result of runaway scripts rather than being intentional. This does not detract from the importance of the phenomenon. On the contrary, situations in which flurries are unintentional add motivation to the need to identify them before using the workload as representative of normal work.

File Server An important generalization of flurries replaces the source component of their definition: instead of being work generated by a single user, we can consider work generated by a singular event. Two such events are shown in Fig. 6.21, displaying a file server's level of activity. The first high-load event, in September 2002, is attributed to a massive copying due to a hardware upgrade. The second, during September to December 2003, is attributed to a bug in a new release

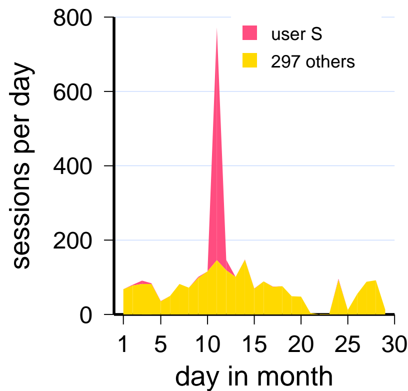


Figure 6.19: *A flurry in the sessions on a Unix server diverts from users' normal working patterns.*

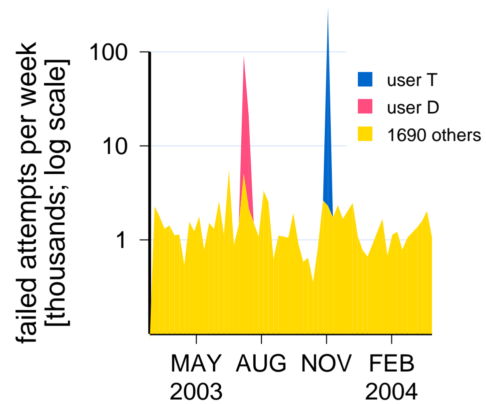


Figure 6.20: *Flurries on an authentication server are 2 order of magnitude bigger than the average.*

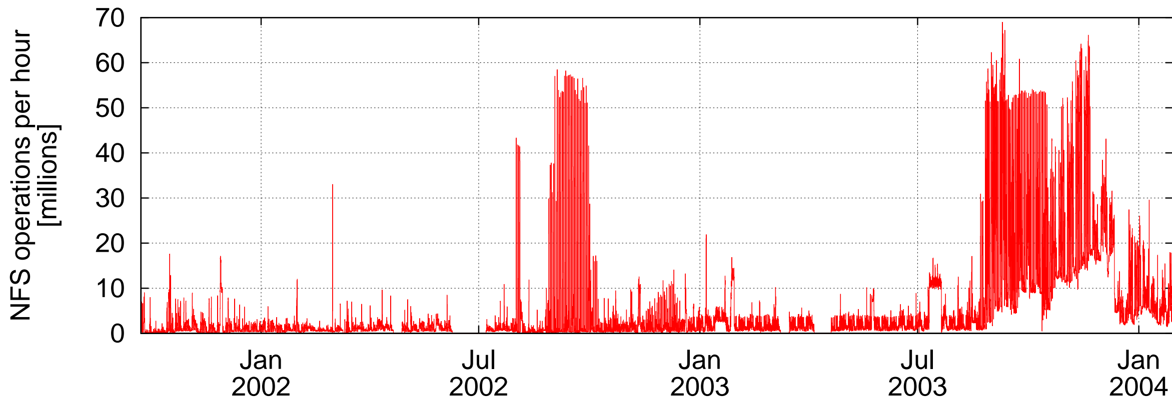


Figure 6.21: *Activity on a departmental NetApp filer.*

of the GNU C library[2] [154]. Installing the new version is the event that triggered this flurry of activity, and fixing it ended the flurry.

Refining the Definition of Flurries There are many accounts of flurry-like events on the Internet, provided we generalize the notion of source from a single user to some singular event that attracts many users (but still a small subset of all Internet users, and for a limited time). For example, new releases of software by Microsoft have caused the so called "midnight madness" phenomenon, where users flocking to download the new version (typically released at midnight) saturate the network and overwhelm the servers [124]. Other examples include the surge of activity on CNN's servers on September 11, 2001, and the usage of sites set up especially to cover sporting events such as the Olympic games or the World Cup finals [5]. All of these events are singular, and lead to unique traffic patterns. We claim that it would be wrong to use workload data including

[2]The bug is that the d_off field in the dirent structure isn't maintained correctly by the auto-mount daemon. Specifically, the 64-bit offset is either 0 or a garbage value. When using a 32-bit file system interface (like the libc readdir routine), getdents verifies that only 32 bits are actually used, and therefore fails if the garbage contains more bits. In trying to handle this error it attempts to seek to the beginning of the erroneous entry, identified using the offset of the previous one. But this is also a garbage value. And if it is 0, we end up with an endless loop of repeatedly reading the first entry, which is what caused the surge of activity seen in Fig. 6.21.

such singular events to analyze the performance of web servers under normal conditions, just as it would be wrong to use normal data for an evaluation of how systems would behave under unique conditions. Of course, in these particular cases, high-load conditions may be more important and meaningful than normal conditions; if this is the case, they should be the focus of study rather than being eliminated as suggested below. For example, Ari et al. model such activity, which they call “flash crowds”, with the aim of evaluating schemes to survive them [4].

Targeted attacks on specific servers also qualify as flurries. In many cases, the nature of the attack is to flood the server and overwhelm it with a load that is much higher than its capacity. This load is generated by a small group of machines (relative to the whole Internet), and lasts for a limited, well-defined time. In this case, an analysis of the attack workload patterns is not only useful for evaluation of servers, but also as a tool in identifying such attacks [11].

6.8 Conclusions

For the conclusion of this chapter, we refer the reader to Section 7.4 (page 127).

Chapter 7

Discussion and Conclusions

The most popular scheduling policy for parallel systems is FCFS with backfilling [53, 37], as introduced by the EASY scheduler [98]. This popularity most probably emanates from the combination of its attractive properties, being

- simple (easy to implement, understand, and maintain),
- fair (uses FCFS as the basis), and
- effective (yields performance results comparable to much more sophisticated algorithms).

The price of these benefits, however, is that users must supply estimates of how long their jobs will run. Estimates are utilized by the system to better pack the jobs by means of exploiting scheduling "holes" to allow short jobs to run ahead of their time, provided they do not delay previously queued jobs (or at least the first). Jobs attempting to exceed their estimates are killed by the system so as not to violate subsequent commitments. Surprisingly, a decade of related studies resulted in an almost overwhelming agreement amongst researchers that inaccurate estimates either do not effect or, more frequently, improve performance [146, 47, 174, 115, 169, 108, 142, 170, 122, 34, 64].

In light of this background, this work has three major contributions. We begin by showing that the "inaccuracy helps" common wisdom is merely an unwarranted artifact of the erroneous manner in which inaccurate estimates have been modeled, and that increased accuracy does in fact improve performance (Section 7.1). We go on to develop a correct model that, from now on, will allow for valid performance evaluations (Section 7.2). We then exploit the new insights and understandings regarding the underlying essence of the workload experienced by parallel systems, to devise a new scheduler that is able to automatically improve the quality of estimates and put this into productive use (Section 7.3). We note that previous attempts to do this [62, 115, 19, 15, 90, 170] yielded algorithms that are inherently different than native backfilling, and that our solution is the first to achieve this goal while preserving all the attractive qualities as listed above.

Finally, a fourth contribution of this work is finding a fundamental flow in the standard methodology of conducting system-related research based on associated production logs: the inclusion of "workload flurries" casts a shadow on the validity of the obtained results. Thus, we propose that workload logs be sanitized to eliminate the problem (Section 7.4).

7.1 Resolving the Misconception of Inaccurate Estimates

The de-facto standard for modeling increasingly inaccurate user estimates has been the f-model that, given a runtime r, uniformly chooses the associated estimate from $[r, r \cdot (f+1)]$ at random,

or deterministically sets it to be $r \cdot (f + 1)$. With this, bigger fs imply increased inaccuracy. The perception that "inaccuracy doesn't affect or improves performance" is largely based on results obtained with this model. Studies reporting a performance improvement explained it with the "holes" argument, claiming that increased overestimation of long jobs opens larger holes in the schedule for backfilling shorter jobs. In contrast, studies reporting performance is unaffected have used the "balance" argument, claiming that larger holes cancel out by the fact backfill candidates appear proportionally longer. While both arguments make sense, they are contradictory, and in any case fail to explain the results as reported below.

We found performance is extremely sensitive to minor changes in f, and that within the noisy results space the two contradictory observations about performance-trends are both possible, when using only few samples in a non-systematic manner. However, averaging over repeated simulations revealed that the mean effect of increasing f is usually V- or L-shaped: in both cases average wait time and slowdown drop at low inaccuracies and then, for V-curves, the trend is gradually reversed for larger fs (though large fs still yield better results than f=0).

To explain this, we show that the seemingly contradictory "balance" and "holes" arguments are both incorrect, or rather, correct to some extent, but miss the key issue that reconciles between them: Performance improvement due to increased f is not simply the result of more backfilling due to more holes in the schedule (in accordance with the "holes" argument), because inflated runtime estimates not only create holes in the schedule, but also enlarge potential backfill jobs, making it harder for them to fit into these holes (in accordance with the "balance" argument). Rather, it is the result of a "heel-and-toe" dynamic: a distinctive sequence of events where small backfill jobs continuously *prevent the holes from closing up*, leading to a preference for short jobs and the automatic production of an SJF-like schedule. When f is very small, the proportionally narrow holes make sure only jobs that are truly short enjoy the effect (explaining the descending part of the V and L-shapes). However, as f gets bigger, increasingly longer jobs can enjoy it too (explaining the ascending part). The situation is worse for the random model, which allows long jobs to masquerade as short and vice versa (explaining why the deterministic model yields better performance and is usually inclined to an L-shape). We have directly quantified this by measuring the "SJFness" as a function of f, defined to be the percent of jobs that are the shortest in the wait-queue at the time they are started. The result was consistently Λ-shaped, a kind of mirror image to the V performance curves. The single L-shaped workload we found (both random and deterministic models) managed to "escape" the reversal of the performance trend due to the fact the activity it embodies lacks temporal burstiness, implying an unpopulated wait-queue and therefore fewer opportunities to mistake long jobs for short. Indeed, when burstiness was artificially added to this log, the L performance curve turned into a V curve.

Importantly, the heel-and-toe effect means that performance improvements due to multiplying the estimates are at the expense of the first queued job, which is repeatedly delayed in favor of shorter/smaller jobs. Directly quantifying this revealed that the "unfairness" of the schedule is proportional to f: the bigger the f, the greater the unfairness. This means that multiplying is simply *trading off fairness for performance* (Fig. 1.10, page 19). In fact, this statement is correct regardless of whether the values being multiplied are actual runtimes (perfect) or were supplied by users (flawed); it's just that the more accurate the initial values we multiply, the better the resulting performance becomes. The bottom line is that *multiplying is actually a scheduling policy*: it is technically possible as well as legitimate for schedulers to multiply the estimates they use, exercising the performance/fairness tradeoff; but users' nature and behavior is completely different,

as will be discussed next.

Fully understanding the f-model highlights its fundamental flaw: it leads to a limited SJF-like scheduling, and indeed, SJF is insensitive to multiplying runtimes by some factor as long as the relative ordering of jobs is preserved. But *real* user estimates provide no such ordering! Rather, as outlined in the next section, they are inherently modal, with 90% of the jobs using only 20 "round" estimate values (e.g. 1 hour) and, in particular, 10-27% using E_{max} — the maximum allowed.[1] Any popular estimate is bad for backfilling as the scheduler cannot tell whether the associated jobs are short or long (e.g. regardless of the estimate, the real runtime is often zero because of the many jobs that fail on startup). However, E_{max} is especially bad, as the associated jobs are never backfilled and thus the more there are jobs that use it, the more the schedule resembles plain FCFS.

We conclude that the popular claim that "increasingly inaccurate estimates improve performance" is only correct if "inaccurate" means "multiplied by a factor", which is far from the truth when real estimates are involved. Inaccuracy of real estimates manifests itself in the form of modality, and "increasing it" means making estimates more modal (e.g. by adjusting the number of jobs associated with E_{max} from 10% to 20%). In this case, *increased inaccuracy actually degrades performance*, as one would intuitively expect. Previous studies that suggested otherwise were simply unaware their results are dominated by the performance/fairness tradeoff. Put in another way, we refute the overwhelmingly accepted myth that inaccuracy improves (or doesn't effect) performance, on the grounds that it is based on false and unrealistic assumptions.

We demonstrate the correctness of our findings by suggesting the *truncated f-model*, which adjusts an estimate e that is generated by the vanilla f-model to be $\min(E_{max}, e)$. This creates a mode at E_{max}, such that bigger fs imply more jobs associated with E_{max}. Indeed, one can "manufacture" arbitrarily bad performance results by choosing a big enough f. Importantly, one can always find an f for which results obtained when using artificial estimates, are equal to those obtained when real estimates are employed, in contrast to the vanilla model. We view the truncated model as a simple "quick and dirty" substitute for the vanilla, and contend it should always be preferred over the latter. Regrettably, the truncated model is still not realistic. For example, it generates only one mode (at E_{max}) and only associates longer jobs with it, whereas with real estimates there are several modes and short jobs are associated with all of them. One consequence was that each trace/metric combination required a significantly different f in order to obtain results comparable to those of real estimates. We therefore advocate the use our accurate estimates model as suggested in the next section.

This part of our work was published in [159].

7.2 Accurately Modeling User Runtime Estimates

While the f-model is the most popular, other estimate models have been suggested. Together they have been used to study the impact of inaccurate estimates on performance (see previous section), and to complement workloads that lacked estimates data [169, 170, 58]. Collectively examining all models, we find each of them to be lacking in some respect. Their shortcomings include implicitly revealing too much information about real runtimes, erroneously emulating the accuracy ratio of runtime to estimate, neglecting to take into consideration the fact that all production installations have a limit on the maximal allowed estimate (E_{max}), and that this value is typically the

[1] Probably due to a combination of the inability of users to accurately predict how long their jobs will run and the strict backfilling policy of killing underestimated jobs.

most popular. Importantly, two key ingredients are missing from existing models: the inherently modal nature of the estimates caused by users' tendency to supply "round" values [108, 17, 93], and the temporal repetitive nature of user estimates, assigning the same value to bursts of jobs (sessions) [173, 133]. The combination of these has a decisive effect on performance results, as low estimate-variance of currently waiting jobs reduces the effectiveness of backfilling. Consequently, the outcome of using the existing models in simulation is invalid performance evaluation results that are unrealistically better than those obtained with real estimates.

Our approach is to develop a model that targets estimates' modality. We view the estimates distribution as a sequence of "modes" (each mode is a pair composed of the estimate's value and the percent of jobs that used it) and investigate their main characteristics. Our findings include the aforementioned invariant that 20 "head" estimates are used by about 90% of the jobs throughout the entire duration of the log. The "popularity" of head estimates (percentage of jobs using them) decreases exponentially, whereas the tail obeys a power-law. The few hundred time values that are used as estimates are well-fitted by a fractional model, while at the same time, 15 out of the 20 head estimates are identical across all the production logs we have examined. The major difficulty we faced was determining how popular is each head estimate (how many jobs are associated with each). This was solved by the "pool algorithm", aimed to capture the many similarities between profiles of head-estimates within the different production logs we analyzed.

We find that *all* modeled aspects of the estimates distribution are almost identical across the logs, and therefore our model defines only two mandatory parameters: the number of jobs and the maximal allowed estimate (E_{max}). While considerable variance does in fact exist, it is mostly encapsulated within the percentage of jobs estimated to run for E_{max} (an optional parameter). The remaining variance (if any) is attributed to another 1-2 very popular modes that sometimes exist, but are unique to individual logs. When provided this additional information, our model produces distributions that are remarkably similar to that of the original. Importantly, the ability of our model to make the resulting distribution more modal through optional parameters, allows for a realistic evaluation of the impact of increasingly inaccurate estimates on performance.

When put to use in simulation (by replacing real estimates with artificial ones), our model consistently yields performance results that are closer to the original than those obtained by other models. In fact, these results are almost identical to when real estimates are used and are randomly shuffled between jobs. This pinpoints the temporal repetitiveness of per-user estimates as the final obstacle separating us from achieving truly realistic results. Future work therefore includes developing an assignment scheme of estimates to jobs that preserves this feature, but this requires the development of a session-based model [148, 130] that is beyond the scope of this work.

Our model can be downloaded from this site [155] within the Parallel Workload Archive [110]. Its interface contains two functions: generating the distribution modes, and assigning estimates to jobs. (The latter is essentially random shuffling of estimates between jobs, under the constraint that runtimes are smaller than estimates.) A utility that makes use of this interface, to append estimates to workloads that are given in Standard Workloads Format [147], is also available for download.

This part of our work was published in [157].

7.3 Leveraging System-Generated Predictions for Backfilling

As noted above, user estimates are inaccurate and modal, a fact that significantly reduces system performance. The alternative is system-generated predictions based on users' history, which are

much more accurate. Despite considerable efforts of researchers [48, 62, 136, 83, 108, 86, 97], predictions were *never* incorporated into production systems. This part of our work is about identifying the problems causing this situation, and providing applicable and easy to use solutions to all of them. Specifically, we identify three major difficulties and thus the contribution of this part of our work is threefold.

The first difficulty is of a technical nature. Under backfilling, user estimates are part of the user contract: jobs that exceed their estimates are killed by the system, so as not to violate subsequent commitments. This makes system-generated predictions unsuitable, as some predictions inevitably turn out too short, and users will not tolerate their jobs being killed prematurely just because of erroneous system speculations. Researchers that noted this problem failed to solve it within the native backfilling framework [62, 115, 108, 15, 90], but our solution is rather simple: (1) use user estimates exclusively as kill-times, (2) base all other scheduling decisions on system-generated predictions, and (3) dynamically increase predictions outlived by their jobs, and push back affected reservations, in order to provide the scheduler with a truthful view of the state of the machine. Applying this to EASY usually results in a ∽25% reduction in average wait time and slowdown. We call this improved algorithm EASY$^+$.

The second major difficulty is related to the common misconception suggesting inaccuracy actually improves performance, and therefore implying that good estimates are "unimportant". As discussed above in great detail, this relies on a number of studies showing significant improvements when deliberately making user estimates even less accurate (e.g. by doubling or randomizing them [174, 115]). In this respect, our contribution has two parts: (1) explaining this phenomenon (Section 7.1), and (2) exploiting it. As noted, doubling helps because it induces "heel and toe" backfilling dynamics that approximates an SJF-like schedule, by repeatedly preventing the first queued job from being started. Thus doubling trades off fairness for performance and should be viewed as a property of the scheduler, not the predictor (indeed, we've shown that the more accurate predictions are, the better the results that doubling obtains). We exploit this new understanding to avoid the performance/fairness tradeoff by explicitly using a shortest job *backfilled first* (SJBF) backfilling order. This leads directly to a performance improvement that was previously incorrectly attributed to doubling, randomizing and other similar stunts. By still preserving FCFS as the basis, we manage to enjoy both worlds: a fair scheduler that nevertheless backfills effectively. Applying this to EASY$^+$ can nearly double the performance (up to 47% reduction in average slowdown). We call this enhanced algorithm EASY^{++}.

The third and final difficulty is related to the usability of previously suggested prediction algorithms. These all suffer from at least one (and sometimes all) of the following drawbacks: (1) they require significant memory and complex data structures to save the history of users, (2) they employ a complicated prediction algorithm (to the point of being off-line), and (3) they pay the price in terms of computational overheads for maintaining the history and searching it [62, 138, 83, 86, 97]. Here too our contribution is twofold: (1) showing that a very simple predictor can do an excellent job, and (2) explaining why. Indeed, the improvements of EASY$^+$ / EASY^{++} reported above were obtained by employing a very simple predictor that is both easy to implement and suffers almost no overheads: the average runtime of the two most recently submitted (and already terminated) jobs by the same user. We have argued that our predictor's success stems from the fact it focuses on *recent* jobs, in contrast to previous predictors that focused on *similar* ones (in terms of various job attributes). This claim is supported by our finding that performance degradation is more or less linearly proportional to the amount of past jobs upon which the prediction is based, suggesting a

prediction window of only one or two jobs is optimal (Fig. 1.19, page 26).

Finally, note that while we focus on improving EASY, we have also shown our techniques can be applied equally well to any other backfilling scheduler. (Indeed, our work has already inspired researchers working on the eNANOS grid [87] to incorporate runtime predictions using our techniques.) The reason we choose to focus on EASY is its popularity in production systems, which may be attributed to the combination of conservative FCFS semantics with improved utilization and performance. Since EASY^{++} essentially preserves these qualities, but consistently outperforms its predecessor in terms of accuracy, predictability, and performance, we believe it has an honest chance to replace EASY as the default configuration of production systems.

This part of our work was published in [156], and is the basis of a pending patent [158].

7.4 Cleaning Workloads From Flurries and Other Anomalies

All the results presented above exclusively rely on the modeling and performance evaluation, through simulation, of activity logs from real production systems. This methodology is standard, and is utilized by numerous computer-systems related papers (e.g. the logs we have used in this work were also extensively used in dozens of other papers [110]). The underlying assumption of this methodology is that recordings of production systems are reliable and representative. We challenge this assumption, demonstrate it is often erroneous, and suggest non-representative anomalies be "sanitized" or "cleaned" from the logs, before they are used.

Beginning with workload characterization and modeling, we note that this activity has been advocated and practiced for many years [55, 1, 12], typically by means of collecting workload traces and creating a statistical model based on fitting the distributions of workload attributes [89]. But such an approach is questionable if the data is not stationary, as seems to be the case in the context of parallel supercomputers: we identify flurries as a specific type of deviations from stationarity that have to be taken into account when creating a workload model.

Continuing with performance evaluations through simulation, we note that this activity is also heavily practiced for many years and constitutes an indispensable tool for system analysts and designers [101]. But when unsanitized workloads (or models based upon them) are utilized as the simulators' input, the results are questionable and might very well be erroneous or misleading. The reason is that real workloads are often "multiclass", meaning they are composed of the "normal" load (that is truly representative of the system being studied), and anomalies (unique and non-representative). The problem is that the latter, less important, part might come to dominate the results of the evaluation, specifically if the anomaly is a "workload flurry": rare surges of activity with a repetitive nature, caused by a single user.

We therefore suggest that a workload be separated into "normal" workload and "flurries". Modeling and the performance evaluation of the normal part can then be performed using current standard methodologies. With modeling, this was shown to significantly promote stationarity, e.g. revealing two halves of the same log initially appearing distinctive, are in fact statistically similar if flurries are removed and only the normal portions are compared. With simulation, this was shown to make results robust to small and insignificant changes applied to the workload, and to enable a clear ranking of alternative system designs, which was unobtainable when utilizing raw logs. Thus, sanitization may be expected to lead to reliable and consistent results that are applicable most of the time (during which flurries are not present). Afterwards, comparing evaluation

results using the cleaned log against those based on the raw log will identify whether the removed flurries actually have a significant effect in the specific case being studied.

The main justification for removing flurries steams from the fact they are rare, unique, and has has an effect during a very short period of the time: Using a workload with a flurry in effect emphasizes the rare and unique event at the expense of normal conditions. Thus *leaving the flurry in* is actually the unjustifiable approach. With respect to performance evaluation, the "flurry removal" can be as subtle as not including the flurry jobs within the average performance metric; but we have shown that the much simpler approach of deleting the flurry from the input altogether has exactly the same effect. To argue for evaluations based on workloads from which flurries are *not* removed, one must argue

1. that the activity of a specific user during a short time should indeed dominate the entire evaluation results,
2. that the evaluation results are valid even though negligible perturbation applied to the workload can significantly change them, possibly swaying them in the opposite direction, and
3. that the results are satisfactory even though they might considerably change if the span of time covered by the evaluation is shifted such that the flurry happens to be excluded.

We speculate most analysts would be reluctant to make such arguments. Likewise, when modeling unsanitized data and fitting an attribute against the raw log, one must be willing to accept the following lose-lose situation: on one hand, results are not representative of the "norm" because they are influenced by the flurries, and do not reflect normal usage; on the other hand results also do not reflect flurries, because flurries have a specific temporal structure (they are concentrated within a limited span of time). In other words, sampling from a distribution that includes a flurry does not produce a flurry; rather, it spreads the flurry evenly over the whole duration of the generated workload. Moreover, any specific flurry is not representative of flurries in general

The question is then how to identify and remove the flurries. The methodology we have used is to plot activity levels as a function of time. In the case of parallel jobs, this means job or process arrivals per unit time. In other contexts, other workload attributes would be appropriate. For example, when analyzing Internet traffic one can tabulate packets and flows; for storage systems, one can look at I/O operations and at bytes transferred.

Once a period of time with exceptionally heavy load is identified, this load should be checked for uniformity and source. The flurries we have identified were all composed of numerous repetitions of the same type of work. Identifying this is the key for removing the flurry from the workload, as the combination of the time frame and the flurry's specific attributes often provide an effective filter. As finding flurries is not trivial, this information should be shared together with the original data. In other words, when workload data is made available, it should be accompanied by all the accumulated knowledge regarding problems with its use, and specifically, with information regarding flurries that occur in it. As a first step, we have added our data to the Parallel Workloads Archive [110], from which our original logs come, and which is used by many researchers for numerous studies of parallel job scheduling.

We view this as a first step because, somewhat surprisingly, computer systems analysts rarely verify the integrity of the data on which they rely for their analysis, and the overwhelmingly common case is to use the data "as is" (e.g. consider all the papers that fit distributions against log files without even considering whether some sanitation is in order [54]). This is in disagreement

with what is routinely done in every statistical analysis, where data is throughly validated, outliers are removed when necessary, etc. Rare studies that do attempt to sanitize, tend to have a "local" or "specific" nature, targeting a single attribute or concept instead of providing a generalization like we do in this part of our work. For example, in an attempt to model the daily cycle of the jobs submittal process, Cirne and Berman clustered days, and excluded clusters populated by only one day from participating in the evaluation [20]. The flurries phenomenon suggests this approach is problematic because (1) "normal" jobs are also needlessly excluded, and (2) flurries may span more than one day and thus be erroneously included. Of course, just eliminating flurries is also not a good solution, as flurries do in fact occur. An open question is how to model or evaluate the effect of the flurries on a system designed and optimized for the more common non-flurry workload. An obvious first step is to use specific flurries that occur in recorded workloads and study their effect. But it is doubtful whether this can predict the effect of other potential flurries. Important future work is therefore to develop methods to extend and generalize the results obtained with specific flurries, and try to derive bounds on the effects of other potential flurries.

To summarize, it is extremely important to use real data regarding the workload on computer systems. But it is equally important to ensure that this is high-quality and representative data. Using measured workloads indiscriminately risks the introduction of unknown anomalies that may lead to unknown effects. Workload flurries are such an anomaly, and should be handled with care.

This part of our work was published in [160, 54].

Bibliography

[1] A. K. Agrawala, J. M. Mohr, and R. M. Bryant, “*An approach to the workload characterization problem*”. *Computer* **9(6)**, pp. 18–32, Jun 1976.

[2] K. Aida, H. Kasahara, and S. Narita, “*Job scheduling scheme for pure space sharing among rigid jobs*”. In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 98–121, Springer Verlag, 1998. Lect. Notes Comput. Sci. vol. 1459.

[3] G. S. Almasi and A. Gottlieb, *Highly Parallel Computing*. Benjamin Cummings Publishing Inc. (Addison Wesley), 2nd ed., 1994.

[4] I. Ari, B. Hong, E. L. Miller, S. A. Brandt, and D. D. E. Long, “*Managing flash crowds on the Internet*”. In 11th *Modeling, Anal. & Simulation of Comput. & Telecomm. Syst.*, pp. 246–249, Oct 2003.

[5] M. Arlitt and T. Jin, “*A workload characterization study of the 1998 world cup web site*”. *IEEE Network* **14(3)**, pp. 30–37, May/Jun 2000.

[6] M. F. Arlitt and C. L. Williamson, “*A synthetic workload model for Internet Mosaic traffic*”. In *Summer Comput. Simulation Conf. (SCSC)*, pp. 852–857, Jul 1995.

[7] R. H. Arpaci, A. C. Dusseau, A. M. Vahdat, L. T. Liu, T. E. Anderson, and D. A. Patterson, “*The interaction of parallel and sequential workloads on a network of workstations*”. In *SIGMETRICS Conf. Measurement & Modeling of Comput. Syst.*, pp. 267–278, May 1995.

[8] J. M. Barton and N. Bitar, “*A scalable multi-discipline, multiple-processor scheduling framework for IRIX*”. In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 45–69, Springer-Verlag, 1995. Lect. Notes Comput. Sci. vol. 949.

[9] A. Batat and D. G. Feitelson, “*Gang scheduling with memory considerations*”. In 14th *Intl. Parallel & Distributed Processing Symp.*, pp. 109–114, May 2000.

[10] A. Bayucan, R. L. Henderson, J. P. Jones, C. Lesiak, B. Mann, B. Nitzberg, T. Proett, and J. Utley, *Portable Batch System Administrator Guide, OpenPBS Release 2.3*. Altair Engineering, Aug 2000. URL http://www.cs.huji.ac.il/labs/parallel/workload/pbs2swf/OpenPBS_AG_2.3.pdf.

[11] M. Burgess, H. Haugerud, S. Straumsnes, and T. Reitan, “*Measuring system normality*”. *ACM Trans. Comput. Syst.* **20(2)**, pp. 125–160, May 2002.

[12] M. Calzarossa and G. Serazzi, “*Workload characterization: a survey*”. *Proc. IEEE* **81(8)**, pp. 1136–1150, Aug 1993.

[13] N. Carriero, E. Freeman, and D. Gelernter, “*Adaptive parallelism on multiprocessors: preliminary experience with Piranha on the CM-5*”. In *Languages and Compilers for Parallel Computing*, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua (eds.), pp. 139–151, Springer-Verlag, Aug 1993. Lect. Notes Comput. Sci. vol. 768.

[14] N. Carriero, E. Freeman, D. Gelernter, and D. Kaminsky, “*Adaptive parallelism and Piranha*”. *Computer* **28(1)**, pp. 40–49, Jan 1995.

[15] S-H. Chiang, A. Arpaci-Dusseau, and M. K. Vernon, “*The impact of more accurate requested runtimes on production job scheduling performance*”. In 8th *Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP)*, D. G. Feitelson, L. Rudolph, and U. Schwiegelshohn (eds.), pp. 103–127, Springer-Verlag, Jul 2002. Lect. Notes Comput. Sci. vol. 2537.

[16] S-H. Chiang, R. K. Mansharamani, and M. K. Vernon, "*Use of application characteristics and limited preemption for run-to-completion parallel processor scheduling policies*". In *SIGMETRICS Conf. Measurement & Modeling of Comput. Syst.*, pp. 33–44, May 1994.

[17] S-H. Chiang and M. K. Vernon, "*Characteristics of a large shared memory production workload*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 159–187, Springer Verlag, 2001. Lect. Notes Comput. Sci. vol. 2221.

[18] S-H. Chiang and M. K. Vernon, "*Dynamic vs. static quantum-based parallel processor allocation*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 200–223, Springer-Verlag, 1996. Lect. Notes Comput. Sci. vol. 1162.

[19] S-H. Chiang and M. K. Vernon, "*Production job scheduling for parallel shared memory systems*". In 15th *IEEE Int'l Parallel & Distributed Processing Symp. (IPDPS)*, Apr 2001.

[20] W. Cirne and F. Berman, "*A comprehensive model of the supercomputer workload*". In 4th *Workshop on Workload Characterization*, Dec 2001.

[21] W. Cirne and F. Berman, "*A model for moldable supercomputer jobs*". In 15th *Intl. Parallel & Distributed Processing Symp.*, Apr 2001.

[22] W. Cirne and F. Berman, "*Using moldability to improve the performance of supercomputer jobs*". *J. of Parallel & Distributed Comput. (JPDC)* **62(10)**, pp. 1571–1601, Oct 2002.

[23] W. Cirne and F. Berman, "*When the herd is smart: aggregate behavior in the selection of job request*". *IEEE Trans. on Parallel & Distributed Syst. (TPDS)* **14(2)**, pp. 181–192, Feb 2003.

[24] Platfrom Computing, *Administering Platform LSF, version 6.2*. Feb 2006. URL www.platform.com/Support/Documentation.htm.

[25] J. Corbalán, X. Martorell, and J. Labarta, "*Performance-driven processor allocation*". In 4th *Symp. Operating Systems Design & Implementation*, pp. 59–71, Oct 2000.

[26] M. E. Crovella, "*Performance evaluation with heavy tailed distributions*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 1–10, Springer Verlag, 2001. Lect. Notes Comput. Sci. vol. 2221.

[27] X. Deng and P. Dymond, "*On multiprocessor system scheduling*". In 8th *Symp. Parallel Algorithms & Architectures*, pp. 82–88, Jun 1996.

[28] X. Deng, N. Gu, T. Brecht, and K. Lu, "*Preemptive scheduling of parallel jobs on multiprocessors*". In 7th *SIAM Symp. Discrete Algorithms*, pp. 159–167, Jan 1996.

[29] J. J. Dongarra, H. W. Meuer, H. D. Simon, and E. Strohmaier, "*Top500 supercomputer sites*". URL http://www.top500.org/. (updated every 6 months).

[30] A. B. Downey, "*A parallel workload model and its implications for processor allocation*". In 6th *Intl. Symp. High Performance Distributed Comput.*, pp. 112–124, Aug 1997.

[31] A. B. Downey, "*Predicting queue times on space-sharing parallel computers*". In 11th *IEEE Int'l Parallel Processing Symp. (IPPS)*, pp. 209–218, Apr 1997.

[32] K. Dussa, K. Carlson, L. Dowdy, and K-H. Park, "*Dynamic partitioning in a transputer environment*". In *SIGMETRICS Conf. Measurement & Modeling of Comput. Syst.*, pp. 203–213, May 1990.

[33] Altair Engineering, *PBS Professional 7.1 Administrator's Guide*. Aug 2005. Editor: Anne Urban. URL http://www.cs.huji.ac.il/labs/parallel/workload/pbs2swf/PBSProAG_7.1.pdf.

[34] D. England, J. Weissman, and J. Sadago-pan, "*A new metric for robustness with application to job scheduling*". In 14th *IEEE Int'l Symp. on High Performance Distributed Comput. (HPDC)*, pp. 135–143, Jul 2005.

[35] C. Ernemann, V. Hamscher, and R. Yahyapour, "*Economic scheduling in grid computing*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson, L. Rudolph, and U. Schwiegelshohn (eds.), pp. 128–152, Springer Verlag, 2002. Lect. Notes Comput. Sci. vol. 2537.

[36] C. Ernemann, M. Krogmann, J. Lepping, and R. Yahyapour, "*Scheduling on the top 50 machines*". In 10th *Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP)*, pp. 17–46, Springer-Verlag, Jun 2004. Lect. Notes Comput. Sci. vol. 3277.

[37] Y. Etsion and D. Tsafrir, *A Short Survey of Commercial Cluster Batch Schedulers*. Technical Report 2005-13, The Hebrew University of Jerusalem, May 2005.

[38] J. J. Evans, C. S. Hood, and C. S. Hood, "*Exploring the relationship between parallel application run-time and network performance in clusters*". In *IEEE Int'l Conf. on Local Comput. Networks (LCN)*, pp. 538–547, Oct 2003.

[39] D. G. Feitelson, "*Experimental analysis of the root causes of performance evaluation results: a backfilling case study*". *IEEE Trans. on Parallel & Distributed Syst. (TPDS)* **16(2)**, pp. 175–182, Feb 2005.

[40] D. G. Feitelson, "*Metric and workload effects on computer systems evaluation*". *Computer* **36(9)**, pp. 18–25, Sep 2003.

[41] D. G. Feitelson, "*Metrics for mass-count disparity*". In *Modeling, Anal. & Simulation of Comput. & Telecomm. Syst.*, pp. 61–68, Sep 2006.

[42] D. G. Feitelson, "*On the interpretation of Top500 data*". *Int'l J. of High Performance Comput. Apps. (IJHPCA)* **13(2)**, pp. 146–153, Summer 1999.

[43] D. G. Feitelson, "*The supercomputer industry in light of the Top500 data*". *IEEE Comput. in Sci. & Eng.* **7(1)**, pp. 42–47, Jan/Feb 2005.

[44] D. G. Feitelson, *A Survey of Scheduling in Multiprogrammed Parallel Systems*. Research Report RC 19790 (87657), IBM T. J. Watson Research Center, Oct 1994.

[45] D. G. Feitelson, A. Batat, G. Benhanokh, D. Er-El, Y. Etsion, A. Kavas, T. Klainer, U. Lublin, and M. A. Volovic, "*The ParPar system: a software MPP*". In *High Performance Cluster Computing, Vol. 1: Architectures and Systems*, R. Buyya (ed.), pp. 754–770, Prentice-Hall, 1999.

[46] D. G. Feitelson and M. A. Jette, "*Improved utilization and responsiveness with gang scheduling*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 238–261, Springer Verlag, 1997. Lect. Notes Comput. Sci. vol. 1291.

[47] D. G. Feitelson and A. Mu'alem Weil, "*Utilization and predictability in scheduling the IBM SP2 with backfilling*". In 12th *IEEE Int'l Parallel Processing Symp. (IPPS)*, pp. 542–546, Apr 1998.

[48] D. G. Feitelson and B. Nitzberg, "*Job characteristics of a production parallel scientific workload on the NASA Ames iPSC/860*". In 1st *Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP)*, D. G. Feitelson and L. Rudolph (eds.), pp. 337–360, Springer-Verlag, Apr 1995. Lect. Notes Comput. Sci. vol. 949.

[49] D. G. Feitelson and L. Rudolph, "*Distributed hierarchical control for parallel processing*". *Computer* **23(5)**, pp. 65–77, May 1990.

[50] D. G. Feitelson and L. Rudolph, "*Evaluation of design choices for gang scheduling using distributed hierarchical control*". *J. Parallel & Distributed Comput.* **35(1)**, pp. 18–34, May 1996.

[51] D. G. Feitelson and L. Rudolph, "*Metrics and benchmarking for parallel job scheduling*". In *Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP)*, D. G. Feitelson and L. Rudolph (eds.), pp. 1–24, Springer-Verlag, Mar 1998. Lect. Notes Comput. Sci. vol. 1459.

[52] D. G. Feitelson and L. Rudolph, "*Toward convergence in job schedulers for parallel supercomputers*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 1–26, Springer-Verlag, 1996. Lect. Notes Comput. Sci. vol. 1162.

[53] D. G. Feitelson, L. Rudolph, and U. Schwiegelshohn, "*Parallel job scheduling — a status report*". In 10th *Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP)*, D. G. Feitelson, L. Rudolph, and U. Schwiegelshohn (eds.), pp. 1–16, Springer-Verlag, Jun 2004. Lect. Notes Comput. Sci. vol. 3277.

[54] D. G. Feitelson and D. Tsafrir, "*Workload sanitation for performance evaluation*". In *IEEE Int'l Symp. Performance Analysis of Syst. & Software (ISPASS)*, pp. 221–230, Mar 2006.

[55] D. Ferrari, "*Workload characterization and selection in computer performance measurement*". *Computer* **5(4)**, pp. 18–24, Jul/Aug 1972.

[56] E. Frachtenberg, D. G. Feitelson, J. Fernandez, and F. Petrini, "*Parallel job scheduling under dynamic workloads*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson, L. Rudolph, and U. Schwiegelshohn (eds.), pp. 208–227, Springer Verlag, 2003. Lect. Notes Comput. Sci. vol. 2862.

[57] E. Frachtenberg, D. G. Feitelson, J. Fernandez-Peinador, and F. Petrini, "*Parallel job scheduling under dynamic workloads*". In *Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP)*, pp. 208–227, Springer-Verlag, 2003. Lect. Notes Comput. Sci. vol. 2862.

[58] E. Frachtenberg, D. G. Feitelson, F. Petrini, and J. Fernandez, "*Adaptive parallel job scheduling with flexible coscheduling*". *IEEE Trans. Parallel & Distributed Syst.* **16(11)**, pp. 1066–1077, Nov 2005.

[59] H. Franke, J. Jann, J. E. Moreira, P. Pattnaik, and M. A. Jette, "*An evaluation of parallel job scheduling for ASCI Blue-Pacific*". In *Supercomputing '99*, Nov 1999.

[60] H. Franke, P. Pattnaik, and L. Rudolph, "*Gang scheduling for highly efficient distributed multiprocessor systems*". In 6th *Symp. Frontiers Massively Parallel Comput.*, pp. 1–9, Oct 1996.

[61] W. Gentzsch, "*Sun grid engine: towards creating a compute power grid*". In *IEEE Int'l Symp. on Cluster Comput. & the Grid (CCGrid)*, pp. 35–36, May 2001.

[62] R. Gibbons, "*A historical application profiler for use by parallel schedulers*". In 3rd *Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP)*, D. G. Feitelson and L. Rudolph (eds.), pp. 58–77, Springer-Verlag, Apr 1997. Lect. Notes Comput. Sci. vol. 1291.

[63] B. Gorda and R. Wolski, "*Time sharing massively parallel machines*". In *Intl. Conf. Parallel Processing*, vol. II, pp. 214–217, Aug 1995.

[64] F. Guim, J. Corbalán, and J. Labarta, *Impact of Qualitative and Quantitative Errors of the Job Runtime Estimation in Backfilling Based Scheduling Policies*. Technical Report, Computer Architecture Department, Technical University of Catalonia (UPC)., 2006. Submitted for publication.

[65] A. Gupta, A. Tucker, and S. Urushibara, "*The impact of operating system scheduling policies and synchronization methods on the performance of parallel applications*". In *SIGMETRICS Conf. Measurement & Modeling of Comput. Syst.*, pp. 120–132, May 1991.

[66] M. W. Hall and M. Martonosi, "*Adaptive parallelism in compiler-parallelized code*". *Concurrency — Pract. & Exp.* **10(14)**, pp. 1235–1250, Dec 1998.

[67] W. Händler, "*Simplicity and flexibility in concurrent computer architecture*". In *High-Speed Computation*, J. S. Kowalik (ed.), pp. 69–88, Springer-Verlag, 1984. NATO ASI Series Vol. F7.

[68] R. L. Henderson, "*Job scheduling under the portable batch system*". In 1st *Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP)*, D. G. Feitelson and L. Rudolph (eds.), pp. 279–294, Springer-Verlag, 1995. Lect. Notes Comput. Sci. vol. 949.

[69] A. Hori, H. Tezuka, and Y. Ishikawa, "*Highly efficient gang scheduling implementation*". In *Supercomputing '98*, Nov 1998.

[70] S. Hotovy, "*Workload evolution on the Cornell Theory Center IBM SP2*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 27–40, Springer-Verlag, 1996. Lect. Notes Comput. Sci. vol. 1162.

[71] C. Isci, G. Contreras, and M. Martonosi, "*Live, runtime phase monitoring and prediction on real systems with application to dynamic power management*". In 39th *IEEE/ACM Int'l Symp. on Microarchit. (MICRO)*, pp. 359–370, Dec 2006.

[72] M. Islam, P. Balaji, P. Sadayappan, and D. K. Panda, "*QoPS: a QoS based scheme for parallel job scheduling*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson, L. Rudolph, and U. Schwiegelshohn (eds.), pp. 252–268, Springer Verlag, 2003. Lect. Notes Comput. Sci. vol. 2862.

[73] N. Islam, A. L. Prodromidis, M. S. Squillante, L. L. Fong, and A. S. Gopal, "*Extensible resource management for cluster computing*". In 17th *Intl. Conf. Distributed Comput. Syst.*, pp. 561–568, May 1997.

[74] D. Jackson, "*Maui/Moab default configuration*". Jan 2006. Personal communication (with CTO of Cluster Resources).

[75] D. Jackson, Q. Snell, and M. Clement, "*Core algorithms of the Maui scheduler*". In 7th *Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP)*, D. G. Feitelson and L. Rudolph (eds.), pp. 87–102, Springer-Verlag, Jun 2001. Lect. Notes Comput. Sci. vol. 2221.

[76] R. Jain, *The Art of Computer Systems Performance Analysis*. John Wiley & Sons, 1991.

[77] J. Jann, P. Pattnaik, H. Franke, F. Wang, J. Skovira, and J. Riodan, "*Modeling of workload in MPPs*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 95–116, Springer Verlag, 1997. Lect. Notes Comput. Sci. vol. 1291.

[78] M. Jette, D. Storch, and E. Yim, "*Timesharing the Cray T3D*". In *Cray User Group*, pp. 247–252, Mar 1996.

[79] J. P. Jones and B. Nitzberg, "*Scheduling for parallel supercomputing: a historical perspective of achievable utilization*". In 5th *Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP)*, D. G. Feitelson and L. Rudolph (eds.), pp. 1–16, Springer-Verlag, Apr 1999. Lect. Notes Comput. Sci. vol. 1659.

[80] L. V. Kalé, S. Kumar, and J. DeSouza, "*A malleable-job system for timeshared parallel machines*". In 2nd *IEEE Int'l Symp. on Cluster Comput. & the Grid (CCGrid)*, p. 230, May 2002.

[81] G. B. Kandiraju and A. Sivasubramaniam, "*Characterizing the d-tlb behavior of spec cpu2000 benchmarks*". In *ACM SIGMETRICS Int'l Conf. on Measurement & Modeling of Comput. Syst.)*, pp. 129–139, Jun 2002.

[82] S. Kannan, M. Roberts, P. Mayes, D. Brelsford, and J. F. Skovira, *Workload Management with LoadLeveler*. IBM, 1st ed., Nov 2001. URL http://www.redbooks.ibm.com/abstracts/sg246038.html.

[83] N. H. Kapadia, J. A. B. Fortes, and C. E. Brodley, "*Predictive application-performance modeling in a computational grid environment*". In 8th *IEEE Int'l Symp. on High Performance Distributed Comput. (HPDC)*, p. 6, Aug 1999.

[84] J. Krallmann, U. Schwiegelshohn, and R. Yahyapour, "*On the design and evaluation of job scheduling algorithms*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 17–42, Springer Verlag, 1999. Lect. Notes Comput. Sci. vol. 1659.

[85] E. Krevat, J. G. Castaños, and J. E. Moreira, "*Job scheduling for the BlueGene/L system*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson, L. Rudolph, and U. Schwiegelshohn (eds.), pp. 38–54, Springer Verlag, 2002. Lect. Notes Comput. Sci. vol. 2537.

[86] S. Krishnaswamy, S. W. Loke, and A. Zaslavsky, "*Estimating computation times of data-intensive applications*". *IEEE Distributed Syst. Online (DS Online)* **5(4)**, Apr 2004.

[87] J. Labarta, J. Corbalán, F. Guim, and I. Rodero, "*The eNANOS grid*". URL http://www.bsc.es/grid/enanos.

[88] J. Labarta, S. Girona, and T. Cortes, "*Analyzing scheduling policies using Dimamas*". *Parallel Comput.* **23(1-2)**, pp. 23–34, Apr 1997.

[89] A. M. Law and W. D. Kelton, *Simulation Modeling and Analysis*. McGraw Hill, 3 ed., 2000.

[90] B. G. Lawson and E. Smirni, "*Multiple-queue backfilling scheduling with priorities and reservations for parallel systems*". In 8th *Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP)*, D. G. Feitelson, L. Rudolph, and U. Schwiegelshohn (eds.), pp. 72–87, Springer-Verlag, Jul 2002. Lect. Notes Comput. Sci. vol. 2537.

[91] B. G. Lawson and E. Smirni, "*Power-aware resource allocation in high-end systems via online simulation*". In 19th *ACM Int'l Conf. on Supercomput. (ICS)*, pp. 229–238, Jun 2005.

[92] B. G. Lawson, E. Smirni, and D. Puiu, "*Self-adapting backfilling scheduling for parallel systems*". In *Int'l Conf. on Parallel Processing (ICPP)*, pp. 593–592, Aug 2002.

[93] C. B. Lee, Y. Schwartzman, J. Hardy, and A. Snavely, "*Are user runtime estimates inherently inaccurate?*". In 10th *Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP)*, D. G. Feitelson, L. Rudolph, and U. Schwiegelshohn (eds.), pp. 253–263, Springer-Verlag, Jun 2004. Lect. Notes Comput. Sci. vol. 3277.

[94] C. B. Lee and A. Snavely, "*On the user-scheduler dialogue: studies of user-provided runtime estimates and utility functions*". *Int'l J. of High Performance Comput. Apps. (IJHPCA)*, 2006. To appear.

[95] C. E. Leiserson, Z. S. Abuhamdeh, D. C. Douglas, C. R. Feynman, M. N. Ganmukhi, J. V. Hill, W. D. Hillis, B. C. Kuszmaul, M. A. St. Pierre, D. S. Wells, M. C. Wong-Chan, S-W. Yang, and R. Zak, "*The network architecture of the Connection Machine CM-5*". *J. Parallel & Distributed Comput.* **33(2)**, pp. 145–158, Mar 1996.

[96] H. Li, D. Groep, J. Templon, and L. Wolters, "*Predicting job start times on clusters*". In 6th *IEEE Int'l Symp. on Cluster Comput. & the Grid (CCGrid)*, May 2004.

[97] H. Li, D. Groep, and L. Walters, "*Workload characteristics of a multi-cluster supercomputer*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson, L. Rudolph, and U. Schwiegelshohn (eds.), pp. 176–193, Springer-Verlag, 2004. Lect. Notes Comput. Sci. vol. 3277.

[98] D. Lifka, "*The ANL/IBM SP scheduling system*". In 1st *Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP)*, D. G. Feitelson and L. Rudolph (eds.), pp. 295–303, Springer-Verlag, Apr 1995. Lect. Notes Comput. Sci. vol. 949.

[99] U. Lublin and D. G. Feitelson, "*The workload on parallel supercomputers: modeling the characteristics of rigid jobs*". *J. Parallel & Distributed Comput.* **63(11)**, pp. 1105–1122, Nov 2003.

[100] W. Ludwig and P. Tiwari, "*Scheduling malleable and nonmalleable parallel tasks*". In 5th *SIAM Symp. Discrete Algorithms*, pp. 167–176, Jan 1994.

[101] M. H. MacDougall, *Simulating Computer Systems: Techniques and Tools*. MIT Press, 1987.

[102] C. McCann, R. Vaswani, and J. Zahorjan, "*A dynamic processor allocation policy for multiprogrammed shared-memory multiprocessors*". *ACM Trans. Comput. Syst.* **11(2)**, pp. 146–178, May 1993.

[103] C. McCann and J. Zahorjan, "*Processor allocation policies for message passing parallel computers*". In *SIGMETRICS Conf. Measurement & Modeling of Comput. Syst.*, pp. 19–32, May 1994.

[104] C. McCann and J. Zahorjan, "*Scheduling memory constrained jobs on distributed memory parallel computers*". In *SIGMETRICS Conf. Measurement & Modeling of Comput. Syst.*, pp. 208–219, May 1995.

[105] E. Medernach, "*Workload analysis of a cluster in a grid environment*". In *Job Scheduling Strategies for Parallel Processing*, Jun 2005.

[106] Sun microsystems, *N1 Grid Engine 6 Administration Guide*. May 2005. URL http://docs.sun.com/app/docs/doc/817-5677.

[107] J. E. Moreira and V. K. Naik, "*Dynamic resource management on distributed systems using reconfigurable applications*". *IBM J. Res. Dev.* **41(3)**, pp. 303–330, May 1997.

[108] A. W. Mu'alem and D. G. Feitelson, "*Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling*". *IEEE Trans. Parallel & Distributed Syst.* **12(6)**, pp. 529–543, Jun 2001.

[109] J. K. Ousterhout, "*Scheduling techniques for concurrent systems*". In 3rd *Intl. Conf. Distributed Comput. Syst.*, pp. 22–30, Oct 1982.

[110] "*Parallel Workloads Archive*". URL http://www.cs.huji.ac.il/labs/parallel/workload/.

[111] K-H. Park and L. W. Dowdy, "*Dynamic partitioning of multiprocessor systems*". *Intl. J. Parallel Programming* **18(2)**, pp. 91–120, Apr 1989.

[112] E. W. Parsons and K. C. Sevcik, "*Benefits of speedup knowledge in memory-constrained multiprocessor scheduling*". *Performance Evaluation* **27&28**, pp. 253–272, Oct 1996.

[113] E. W. Parsons and K. C. Sevcik, "*Coordinated allocation of memory and processors in multiprocessors*". In *SIGMETRICS Conf. Measurement & Modeling of Comput. Syst.*, pp. 57–67, May 1996.

[114] E. Perelman, G. Hamerly, M. V. Biesbrouck, T. Sherwood, and B. Calder, "*Using SimPoint for accurate and efficient simulation*". In *ACM SIGMETRICS Int'l Conf. on Measurement & Modeling of Comput. Syst.)*, pp. 318–319, Jun 2003.

[115] D. Perkovic and P. J. Keleher, "*Randomization, speculation, and adaptation in batch schedulers*". In *ACM/IEEE Supercomputing (SC)*, p. 7, Sep 2000.

[116] J. Pruyne and M. Livny, "*Parallel processing on dynamic resources with CARMI*". In 1st *Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP)*, D. G. Feitelson and L. Rudolph (eds.), pp. 259–278, Springer-Verlag, Apr 1995. Lect. Notes Comput. Sci. vol. 949.

[117] M. K. Qureshi, M. A. Suleman, and Y. N. Patt, "*Line distillation: increasing cache capacity by filtering unused words in cache lines*". In *IEEE Int'l Symp. on High-Performance Comput. Archit. (HPCA)*, pp. 250–259, Feb 2007.

[118] Cluster Resources, *Moab Wokload Manager Administrator's Guide*. 2006. Version 4.5.0. URL http://www.clusterresources.com/moabdocs/MoabAdminGuide450.pdf.

[119] E. Rosti, G. Serazzi, E. Smirni, and M. S. Squillante, "*The impact of I/O on program behavior and parallel scheduling*". In *SIGMETRICS Conf. Measurement & Modeling of Comput. Syst.*, pp. 56–65, Jun 1998.

[120] E. Rosti, E. Smirni, L. W. Dowdy, G. Serazzi, and B. M. Carlson, "*Robust partitioning schemes of multiprocessor systems*". *Performance Evaluation* **19(2-3)**, pp. 141–165, Mar 1994.

[121] E. Rosti, E. Smirni, G. Serazzi, and L. W. Dowdy, "*Analysis of non-work-conserving processor partitioning policies*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 165–181, Springer-Verlag, 1995. Lect. Notes Comput. Sci. vol. 949.

[122] G. Sabin and P. Sadayappan, "*On enhancing the reliability of job schedulers*". In *High Availability & Performace Computing Workshop (HAPCW)*, Oct 2005.

[123] G. Sabin and P. Sadayappan, "*Unfairness metrics for space-sharing parallel job schedulers*". In 11th *Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP)*, D. G. Feitelson, E. Frachtenberg, L. Rudolph, and U. Schwiegelshohn (eds.), pp. 238–256, Springer-Verlag, Jun 2005. Lect. Notes Comput. Sci. vol. 3834.

[124] E. Schooler and J. Gemmel, *Using multicast FEC to solve the midnight madness problem*. Technical Report MS-TR-97-25, Microsoft Research, Sep 1997.

[125] B. Schroeder and M. Harchol-Balter, "*Evaluation of task assignment policies for supercomputing servers: the case for load unbalancing and fairness*". In *IEEE Int'l Symp. on High Performance Distributed Comput. (HPDC)*, p. 211, Aug 2000.

[126] S. Setia, M. S. Squillante, and V. K. Naik, "*The impact of job memory requirements on gang-scheduling performance*". *Performance Evaluation Rev.* **26(4)**, pp. 30–39, Mar 1999.

[127] S. K. Setia, "*The interaction between memory allocation and adaptive partitioning in message-passing multicomputers*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 146–164, Springer-Verlag, 1995. Lect. Notes Comput. Sci. vol. 949.

[128] S. K. Setia, M. S. Squillante, and S. K. Tripathi, "*Processor scheduling on multiprogrammed, distributed memory parallel computers*". In *SIGMETRICS Conf. Measurement & Modeling of Comput. Syst.*, pp. 158–170, May 1993.

[129] S. Shankland, "*Power could cost more than servers, Google warns*". CNET news.com, URL http://news.com.com/2100-1010_3-5988090.html, Dec 2005.

[130] E. Shmueli, *Leveraging feedback in evaluating parallel system schedulers*. PhD thesis, The Hebrew University of Jerusalem, Israel, 200? In preparation.

[131] E. Shmueli and D. G. Feitelson, "*Backfilling with lookahead to optimize the packing of parallel jobs*". *J. of Parallel & Distributed Comput. (JPDC)* **65(9)**, pp. 1090–1107, Sep 2005.

[132] E. Shmueli and D. G. Feitelson, "*Backfilling with lookahead to optimize the performance of parallel job scheduling*". In 9th *Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP)*, D. G. Feitelson, L. Rudolph, and U. Schwiegelshohn (eds.), pp. 228–251, Springer-Verlag, Jun 2003. Lect. Notes Comput. Sci. vol. 2862.

[133] E. Shmueli and D. G. Feitelson, "*Using site-level modeling to evaluate the performance of parallel system schedulers*". In *Modeling, Anal. & Simulation of Comput. & Telecomm. Syst.*, pp. 167–176, Sep 2006.

[134] E. Smirni, E. Rosti, G. Serazzi, L. W. Dowdy, and K. C. Sevcik, "*Performance gains from leaving idle processors in multiprocessor systems*". In *Intl. Conf. Parallel Processing*, vol. III, pp. 203–210, Aug 1995.

[135] K. A. Smith and M. I. Seltzer, "*File system aging—increasing the relevance of file system benchmarks*". In *SIGMETRICS Conf. Measurement & Modeling of Comput. Syst.*, pp. 203–213, Jun 1997.

[136] W. Smith, I. Foster, and V. Taylor, "*Predicting application run times using historical information*". In 4th *Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP)*, D. G. Feitelson and L. Rudolph (eds.), pp. 122–142, Springer-Verlag, Mar 1998. Lect. Notes Comput. Sci. vol. 1459.

[137] W. Smith, I. Foster, and V. Taylor, "*Scheduling with advanced reservations*". In 14th *IEEE Int'l Parallel & Distributed Processing Symp. (IPDPS)*, pp. 127–132, May 2000.

[138] W. Smith, V. Taylor, and I. Foster, "*Using run-time predictions to estimate queue wait times and improve scheduler performance*". In 5th *Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP)*, D. G. Feitelson and L. Rudolph (eds.), pp. 202–219, Springer-Verlag, Apr 1999. Lect. Notes Comput. Sci. vol. 1659.

[139] Q. O. Snell, M. J. Clement, and D. B. Jackson, "*Preemption based backfill*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson, L. Rudolph, and U. Schwiegelshohn (eds.), pp. 24–37, Springer Verlag, 2002. Lect. Notes Comput. Sci. vol. 2537.

[140] M. S. Squillante, "*On the benefits and limitations of dynamic partitioning in parallel computer systems*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 219–238, Springer-Verlag, 1995. Lect. Notes Comput. Sci. vol. 949.

[141] S. Srinivasan, R. Kettimuthu, V. Subramani, and P. Sadayappan, "*Selective reservation strategies for backfill job scheduling*". In 8th *Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP)*, D. G. Feitelson, L. Rudolph, and U. Schwiegelshohn (eds.), pp. 55–71, Springer-Verlag, Jul 2002. Lect. Notes Comput. Sci. vol. 2537.

[142] S. Srinivasan, R. Kettimuthu, V. Subrarnani, and P. Sadayappan, "*Characterization of backfilling strategies for parallel job scheduling*". In *Int'l Conf. on Parallel Processing (ICPP)*, pp. 514–522, Aug 2002.

[143] "*Standard performance evaluation corporation*". URL http://www.spec.org.

[144] A. Streit, "*A self-tuning job scheduler family with dynamic policy switching*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson, L. Rudolph, and U. Schwiegelshohn (eds.), pp. 1–23, Springer Verlag, 2002. Lect. Notes Comput. Sci. vol. 2537.

[145] V. Subramani, R. Kettimuthu, S. Srinivasan, and P. Sadayappan, "*Distributed job scheduling on computational grids using multiple simultaneous requests*". In 11th *IEEE Int'l Symp. on High Performance Distributed Comput. (HPDC)*, p. 359, Jul 2002.

[146] T. Suzuoka, J. Subhlok, and T. Gross, *Evaluating Job Scheduling Techniques for Highly Parallel Computers*. Technical Report CMU-CS-95-149, School of Computer Science, Carnegie Mellon University, Aug 1995.

[147] "*The standard workload format (SWF*". URL http://www.cs.huji.ac.il/labs/parallel/workload/swf.html.

[148] D. Talby, *User Modeling of Parallel Workloads*. PhD thesis, The Hebrew University of Jerusalem, Israel, 200? In preparation.

[149] D. Talby and D. G. Feitelson, “*Improving and stabilizing parallel computer performance using adaptive scheduling*”. In 19th *Intl. Parallel & Distributed Processing Symp.*, Apr 2005.

[150] D. Talby and D. G. Feitelson, “*Supporting priorities and improving utilization of the IBM SP scheduler using slack-based backfilling*”. In 13th *Intl. Parallel Processing Symp.*, pp. 513–517, Apr 1999.

[151] D. Talby, D. G. Feitelson, and A. Raveh, “*A co-plot analysis of logs and models of parallel workloads*”. *ACM Trans. on Modeling & Comput. Simulation (TOMACS)*, 2007. To apper.

[152] 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, The Hebrew University of Jerusalem, Aug 2006.

[153] P. Terry, A. Shan, and P. Huttunen, “*Improving application performance on HPC systems with process synchronization*”. *Linux Journal* **2004(127)**, pp. 68–73, Nov 2004. URL http://portal.acm.org/citation.cfm?id=1029015.1029018.

[154] D. Tsafrir, “*Bug (+fix) in getdents() [glibc-2.3.2/linux-2.4.22/i686]*”. URL http://sources.redhat.com/ml/bug-glibc/2003-12/msg00028.html, Dec 2003.

[155] D. Tsafrir, Y. Etsion, , and D. G. Feitelson, “*A model/utility for generating user runtime estimates and appending them to a standard workload format (SWF) file*”. URL http://www.cs.huji.ac.il/labs/parallel/workload/m_tsafrir05, Feb 2006.

[156] D. Tsafrir, Y. Etsion, and D. G. Feitelson, “*Backfilling using system-generated predictions rather than user runtime estimates*”. *IEEE Trans. on Parallel & Distributed Syst. (TPDS)*, 2007. To apper.

[157] D. Tsafrir, Y. Etsion, and D. G. Feitelson, “*Modeling user runtime estimates*”. In 11th *Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP)*, D. G. Feitelson, E. Frachtenberg, L. Rudolph, and U. Schwiegelshohn (eds.), pp. 1–35, Springer-Verlag, Jun 2005. Lect. Notes Comput. Sci. vol. 3834.

[158] D. Tsafrir, Y. Etsion, D. Talby, and D. G. Feitelson, “*System and method for backfilling with system-generated predictions rather than user runtime estimates*”. Patent Application PCT/IL2006/000199, Feb 2006. Pending.

[159] D. Tsafrir and D. G. Feitelson, “*The dynamics of backfilling: solving the mystery of why increased inaccuracy may help*”. In 2nd *IEEE Int’l Symp. on Workload Characterization (IISWC)*, Oct 2006.

[160] D. Tsafrir and D. G. Feitelson, “*Instability in parallel job scheduling simulation: the role of workload flurries*”. In 20th *IEEE Int’l Parallel & Distributed Processing Symp. (IPDPS)*, p. 10, Apr 2006.

[161] D. Tsafrir, K. Ouaknine, and D. G. Feitelson, *Reducing Performance Evaluation Sensitivity and Variability by Input Shaking*. Technical Report 2007-24, School of Computer Science and Engineering, the Hebrew University of Jerusalem, May 2007. Submitted.

[162] G. Utrera, J. Corbalán, and J. Labarta, “*Another approach to backfilled jobs: applying virtual malleability to expired windows*”. In 19th *Intl. Conf. Supercomputing*, pp. 313–322, Jun 2005.

[163] S. Vasupongayya, S-H. Chiang, and B. Massey, “*Search-based job scheduling for parallel computer workloads*”. In *IEEE Int’l Conf. on Cluster Comput. (Cluster)*, Sep 2005.

[164] R. Vaswani and J. Zahorjan, “*The implications of cache affinity on processor scheduling for multiprogrammed, shared memory multiprocessors*”. In 13th *Symp. Operating Systems Principles*, pp. 26–40, Oct 1991.

[165] M. Wan, R. Moore, G. Kremenek, and K. Steube, “*A batch scheduler for the Intel Paragon with a non-contiguous node allocation algorithm*”. In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 48–64, Springer-Verlag, 1996. Lect. Notes Comput. Sci. vol. 1162.

[166] J. William A. Ward, C. L. Mahood, and J. E. West, “*Scheduling jobs on parallel systems using a relaxed backfill strategy*”. In 8th *Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP)*, pp. 103–127, Springer-Verlag, Jul 2002. Lect. Notes Comput. Sci. vol. 2537.

[167] C. Yu and C. R. Das, "*Limit allocation: an efficient processor management scheme for hypercubes*". In *Intl. Conf. Parallel Processing*, vol. II, pp. 143–150, Aug 1994.

[168] K. K. Yue and D. J. Lilja, "*Loop-level process control: an effective processor allocation policy for multiprogrammed shared-memory multiprocessors*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 182–199, Springer-Verlag, 1995. Lect. Notes Comput. Sci. vol. 949.

[169] Y. Zhang, H. Franke, J. Moreira, and A. Sivasubramaniam, "*Improving parallel job scheduling by combining gang scheduling and backfilling techniques*". In 14th *IEEE Int'l Parallel & Distributed Processing Symp. (IPDPS)*, pp. 133–142, May 2000.

[170] Y. Zhang, H. Franke, J. Moreira, and A. Sivasubramaniam, "*An integrated approach to parallel scheduling using gang-scheduling, backfilling, and migration*". *IEEE Trans. on Parallel & Distributed Syst. (TPDS)* **14**(**3**), pp. 236–247, Mar 2003.

[171] Y. Zhang, H. Franke, J. E. Moreira, and A. Sivasubramaniam, "*An integrated approach to parallel scheduling using gang-scheduling, backfilling, and migration*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 133–158, Springer Verlag, 2001. Lect. Notes Comput. Sci. vol. 2221.

[172] S. Zhou, X. Zheng, J. Wang, and P. Delisle, "*Utopia: a load sharing facility for large, heterogeneous distributed computer systems*". *Software — Pract. & Exp. (SPE)* **23**(**12**), pp. 1305–1336, Dec 1993.

[173] J. Zilber, O. Amit, and D. Talby, "*What is worth learning from parallel workloads? A user and session based analysis*". In 19th *Intl. Conf. Supercomputing*, pp. 377–386, Jun 2005.

[174] D. Zotkin and P. J. Keleher, "*Job-length estimation and performance in backfilling schedulers*". In 8th *IEEE Int'l Symp. on High Performance Distributed Comput. (HPDC)*, p. 39, Aug 1999.