Distribution-based cluster scheduling

Jun Woo Park

CMU-CS-19-107
May 2019

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Gregory R. Ganger, Chair
Phillip B. Gibbons
George Amvrosiadis
Michael Kozuch, Intel Labs

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy.

Copyright © 2019 Jun Woo Park

This research was sponsored by the National Science Foundation under grant number IIS-1409802, the U.S. Army Research Office under grant number DAAD190210389, Intel ISTC-CC, Intel Big Data, Intel ISTC-VCC, and a Samsung Scholarship.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

Keywords: Planning under uncertainty, Cluster Scheduling, Cloud Computing

For Sun Hee, Toffee, Coco, and the yet-to-be-born member of the family.

Abstract

Modern computing clusters support a mixture of diverse activities, ranging from customer-facing internet services, software development and test, scientific research, and exploratory data analytics. Many schedulers exploit knowledge of pending jobs' runtimes and resource usages as a powerful building block but suffer significant performance penalty if such knowledge is imperfect. This dissertation demonstrates that schedulers that rely on information about job runtimes and resource usages can more robustly address imperfect predictions by looking at likelihoods of possible outcomes rather than single point expected outcomes.

This dissertation presents a workload analysis and two case studies of scheduling systems: 3Sigma and DistSched. Characterization of real workloads revealed that there exists inherent variability in the job runtimes and resource usage that cannot be captured by single point estimates. An evaluation of a history-based runtime predictor with four different traces demonstrates it is not trivial to obtain perfect runtime predictions in real workloads, especially if the predictor is provided with insufficient information. 3Sigma is a scheduler that leverages distributions of the relevant runtime histories rather than just a point estimate derived from it. By leveraging distribution and mis-estimate mitigation mechanisms, 3Sigma is able to make more robust scheduling decisions and outperform state-of-the-art scheduling systems that only rely on limited or no runtime knowledge. DistSched is a scheduler that leverages distribution of the resource usage (cpu, memory, and cpu-time) and account for the risk of contention to make robust scheduling decisions. The evaluation of DistSched demonstrates that leveraging full history and mitigation mechanisms allows the scheduler to more robustly address the imperfect predictions and perform almost as good as the hypothetical system equipped with perfect knowledge of runtime and resource usage.

Acknowledgments

I have met and got help from so many people during the course of my Ph.D. study and I owe my success in completing the program and getting the degree to them.

The most important person I need to attribute is my advisor Greg Ganger. Through his directions and help, I was able to grow as a researcher. He is the person who made my ramblings to become well-baked goods. Many of the ideas present in the dissertation wouldn't exist without his guidance. I feel very fortunate to be advised by him and he will always be my role model going forward.

The next person in line is Michael Kozuch, who I consider as an unofficial co-advisor. He and I worked together on most projects since I started working with Greg and his feedback was equally crucial in developing ideas present in this dissertation.

I would also like to thank the rest of the members of my thesis committee, Phil Gibbons and George Amvrosiadis, for their valuable insights and feedback.

Throughout the course of my Ph.D., I have collaborated with many different people and appreciated insightful discussion with them. I am very fortunate to work with Alexey Tumanov and Timothy Zhu who were the students who started the original cluster scheduling project, who have helped me to get up to speed with the cluster scheduling research. Angela Jiang and I worked together a lot in the estimator part, both in JamaisVu and eventually in 3Sigma.

The Parallel Data Laboratory (PDL) is an absolutely fabulous group of faculty and students. The amount of feedback and suggestions we get from the weekly meetings, annual retreats, and visit days and the amount of support and opportunities we get just by being part of the group is unfathomable. I specially thank Bill Courtright, Garth Gibson, Majd Sakr, and Andy Pavlo, as well as students, Abutalib Aghayev, Joy Arulraj, Rachata Ausavarungnirun, Ben Blum, Christopher Canel, Andrew Chung, Henggang Cui, Chris Fallin, Aaron Harlap, Kevin Hsieh, Saurabh Kadekodi, Rajat Kateja, Jin Kyu Kim, Michael Kuchnik, Hyeontaek Lim, Charles McGuffey, Jinliang Wei, and Daniel Wong.

I also appreciate the support from an amazing group of staff at PDL: Karen Lindenfelser for streamlining administrative affairs and organizing PDL events, Joan Digney for helping us tremendously in making impressive posters, Chuck Cranor, Chad Dougherty, Mitch Franzos, Jason Boles, and Xiaolin Zang for providing wonderful technical support for the PDL computing resources. I also appreciate the support from the staffs of the Computer Science Department, notably Deb Cavlovich who wields magical powers to make things work and allowed us to focus solely on our research and the Ph.D. program. They have made my research and life as a graduate student a lot easier.

I also want to thank my friends and colleagues. I thank my colleagues at Computer Science Department, Kiryong Ha, Junchen Jiang, Hanbyul Joo, Gunhee Kim, Jisu Kim, Soonho Kong, Euiwoong Lee, Jay-yoon Lee, Seunghak Lee, Mu Li, Vittorio Perera, Kijung Shin, and Manzil Zaheer for providing valuable insights and feedback. I also thank my friends for emotional support and being part of my memorable time in Pittsburgh. There are too many to list, but to name a few, I would like to thank Se-Joon

Chung, SooHyun Jeon, Minkyung Kang, Daehyeok Kim, Dohyeon Kim, Chloe Kim, Jihee Kim, Joshua Kwangho Kim, Taekyun Kim, Gihyuk Ko, Kate Seokjeong Lee, Kiwan Maeng, Shane Moon, Soojin Moon, Diana Nam, and Daegun Won. I also thank my friends from elsewhere, Ilhwang Cha, Dongho Chang, Donghyun Choi, Han Choi, Sungkwon Hong, Daniel Kim, Hojin Kim, Jaehwan Kim, Eunsoo Lee, Jangjik Lee, Sanghyun Lee, Changho Oh, and Alex Park.

I also thank the members of the PDL Consortium: Alibaba, Amazon, Datrium, Dell EMC, Facebook, Google, Hewlett Packard Enterprise, Hitachi Ltd., IBM Research, Intel Corporation, Micron, Microsoft Research, NetApp, Inc., Oracle Corporation, Salesforce, Samsung Semiconductor Inc., Seagate Technology, Two Sigma, Veritas and Western Digital for their interest, insights, feedback, and support. I also would like to thank Los Alamos National Laboratory and Two Sigma for their data, feedback, and support. This research was sponsored by the National Science Foundation under grant number IIS-1409802, the U.S. Army Research Office under grant number DAAD190210389, Intel ISTC-CC, Intel Big Data, Intel ISTC-VCC, and a Samsung Scholarship. Besides the scholarship, I am especially grateful to the staff of the Samsung Scholarship, Yongnyun Kim, Junghyun Kim, Seyoung Na, Jiyeon Park, and Hyunmo Yoo for their support.

Lastly, I want to thank my wife, my closest friend, who I met during early days of my grad school and who was always with me including the brightest and darkest times of my Ph.D. program. I also thank my parents, my brother, Toffee, and Coco for supporting me during the study.

Contents

1 Introduction 1
1.1 Thesis Statement . . . 2
1.2 Contributions . . . 3
1.3 Outline . . . 4

2 Background 5
2.1 Resource consolidation . . . 5
2.2 Cluster scheduling with job information . . . 6
2.3 Predicting job information . . . 6
2.4 Workloads used in this dissertation . . . 7
  2.4.1 Google cluster . . . 7
  2.4.2 Two Sigma clusters . . . 8
  2.4.3 LANL Mustang cluster . . . 9
  2.4.4 LANL OpenTrinity supercomputer . . . 9

3 Diversity of cluster workloads 11
3.1 Dataset information . . . 12
3.2 Job characteristics . . . 12
3.3 Workload heterogeneity . . . 15
3.4 Resource utilization . . . 17
3.5 Failure analysis . . . 19
3.6 A case study on plurality and diversity . . . 23
  3.6.1 JVuPredict background . . . 23
  3.6.2 Evaluation results . . . 24
3.7 On the importance of trace length . . . 26
3.8 Related work . . . 27
3.9 Conclusion . . . 27

4 3Sigma: a runtime distribution based scheduler 29
4.1 Background and related work . . . 31
  4.1.1 Runtime variation and uncertainty . . . 31
  4.1.2 Mis-estimate mitigation strategies . . . 32
  4.1.3 Distribution-based scheduling . . . 33
4.2 Distribution-based Scheduling . . . 34

4.2.1 Valuation of scheduling options 34
4.2.2 Expected resource consumption 35
4.3 Design and implementation 35
4.3.1 Generating runtime distributions 36
4.3.2 Handling imperfect distributions 37
4.3.3 Scheduling algorithm 38
4.4 Evaluation 43
4.4.1 Experimental setup 43
4.4.2 End-to-end performance 46
4.4.3 Attribution of benefit 48
4.4.4 Distribution-based scheduling benefits 48
4.4.5 Sensitivity analyses 49
4.4.6 Scalability 51
4.5 Summary 51

5 DistSched: a resource-runtime distribution based scheduler 53
5.1 Background 54
5.1.1 Predictability 55
5.1.2 Mitigation strategies 56
5.1.3 Assumptions 57
5.2 Resource distribution-based scheduling 58
5.2.1 Valuation of scheduling options 59
5.2.2 Scheduling challenges 60
5.2.3 Greedy scheduling algorithm 62
5.3 Implementation 64
5.3.1 `DSPredict` 65
5.3.2 Mis-estimate mitigation strategies 65
5.4 Evaluation 66
5.4.1 Experimental setup 66
5.4.2 End-to-end performance 69
5.4.3 Benefit attribution 71
5.4.4 Sensitivity to the cluster size 73
5.4.5 Scheduler scalability 73
5.5 Summary 74

6 Conclusion 77
6.1 Future work 77
6.1.1 Making better use of current resource observation 78
6.1.2 Exploiting patterns of resource usage 78
6.1.3 Dependency-aware scheduling 78
6.1.4 Public Cloud or Hybrid Cloud environments 78
6.1.5 Greedy scheduling algorithms 78
6.1.6 Utility functions 79
6.1.7 Adapting to drift and trend in the history 79

Bibliography **81**

List of Figures

3.1 CDF of job sizes based on allocated CPU cores. 13

3.2 CDF of the durations of individual jobs. 14

3.3 Hourly job submission rates for a given day. The lines represent the median, while the shaded region shows the distance between the 25^{th} and 75^{th} percentiles. . . . 15

3.4 Hourly task placement requests for a given day. The lines represent the median, while the shaded region shows the distance between the 25^{th} and 75^{th} percentiles. 16

3.5 CDF of job interarrival times. 18

3.6 CDF of the number of tasks per job. 19

3.7 Breakdown of the total number of jobs, as well as CPU time, by job outcome. . . 20

3.8 CDFs of job sizes (in CPU cores) for unsuccessful and successful jobs. 21

3.9 Success rates for jobs grouped by CPU hours. 22

3.10 Accuracy of JVuPredict predictions of runtime estimates, for all four traces. . . . 24

3.11 *Is a month representative of the overall workload?* The boxplots show distributions of the average job inter-arrival period (left) and duration (right) per month, normalized by the trace's overall average. Boxplot whiskers are defined at 1.5 times the distribution's Inter-Quartile Range (standard Tukey boxplots). 25

4.1 Comparison of 3Sigma with three other scheduling approaches w.r.t. SLO (deadline) miss rate, for a mix of SLO and best effort jobs derived from the Google cluster trace [72] on a 256-node cluster. (Details in §4.4.1) 3Sigma, despite estimating runtime distributions online with imperfect knowledge of job classification, approaches the performance of a hypothetical scheduler using perfect runtime estimates (`PointPerfEst`). Full historical runtime distributions and mis-estimation handling helps 3Sigma outperform `PointRealEst`, a state-of-the-art point-estimate-based scheduler (detailed in §4.1.2). The value of exploiting runtime information, when done well, is confirmed by comparison to a conventional priority-based approach (`Prio`). 30

4.2 Analyses of cluster workloads from three different environments: (a) Distribution of job runtimes (b) Distribution of Coefficient of Variation for each subset grouped by user id (c) Distribution of Coefficient of Variation for each subset grouped by amount of resources requested (d) Histogram of Estimate Errors comparing runtime estimates from the state-of-the-art JVuPredict predictor and actual job runtimes. Estimate Error values computed by $\frac{\text{estimate}-\text{actual}}{\text{actual}} \times 100$. Each datapoint is a bucket representing values within 5% of the nearest decile. The "tail" datapoint includes all estimate errors $> 95\%$. Cluster:SC. Workload:Google_E2E, TwoSigma_E2E, MUSTANG_E2E 32

4.3 Example curves for estimating utility for a given job. Each job is associated with a utility function (a) describing its value as a function of completion time. 3σPredict produces a PDF (b) describing potential runtimes for the job. 3σSched combines them to compute expected utility (c) for the job as a function of its start time. [Note the different x-axes for (a), (b), and (c).] As described in §4.3.2, the overestimate handling technique involves modifying the utility function (a) associated with the job with an extended version illustrated in (d). 34

4.4 End-to-end system integration. 36

4.5 Job D is an SLO job with a 15min deadline. Job BE is a BE job. Left column: job runtimes $\sim U(0,10)$ (scenario1). Right column: job runtimes $\sim U(2.5,7.5)$ with the same $\mu = 5$ (scenario 2). (a) and (b): The final order that yields maximal utility. The intensity of the black represents the expected resource consumption at the start of each slot (from 100% certainty (darkest) to 25% certainty (lightest), in 25% decrements for the scenario 1 and a 50% decrement for the scenario 2. (c) and (d): Inverse CDF $(1-CDF(t))$, the probability of D (blue) and BE (red) jobs completing before t, which is also the probability of still using the resource at that time. (e) and (f): SLO job's expected utility, set to the probability of the job's completion by the deadline at each start time in this example (note: x-axis is different from other subfigs). 40

4.6 Compares the performance of 3Sigma with other systems in the real cluster. 3Sigma constantly outperforms `PointRealEst` and `Prio` on SLO miss-rate and Goodput while nearly matching `PointPerfEst`. Cluster:RC256. Workload:E2E 46

4.7 Compares the performance of 3Sigma with other systems under workloads from different environments in simulated cluster. 3Sigma constantly outperforms `PointRealEst` and `Prio` on SLO miss rate and Goodput while nearly matching `PointPerfEst`. The Google workload is 5hr variant of E2E. Cluster:SC256. Workload:E2E, TwoSigma_E2E, MUSTANG_E2E 47

4.8 **Attribution of Benefit.** The lines representing 3Sigma with individual techniques disabled— demonstrating that all are needed to achieve the best performance. The workload is E2E with a constant deadline slack. Cluster:SC256 Workload:DEADLINE-n where $n \in [20,40,60,80,100,120,140,160,180]$ 48

4.9 3Sigma's performance when artificially varying runtime distribution shift (x-axis) and width (Coefficient of Variation curves). The runtime distribution provided to the scheduler is $\sim \mathcal{N}(\mu = job\_runtime * (1+\frac{x}{100}), \sigma = job\_runtime * CoV)$. Each trace consists of jobs that are either within 10% accuracy or under- or over-estimates jobs. The group of jobs achieves a target average artifical shift. (c) shows the breakdown of these job types for each artificial shift value. Distribution-based schedulers always outperforms the point estimate-based scheduler. Tighter distributions perform better than wider distribution with a smaller artificial shift, but wider distributions are better with a larger artificial shift. The workload is 2 hrs in length. Cluster:SC256. Workload:E2E 49

4.10 3Sigma outperforms others on SLO misses for a range of loads, matching `PointPerfEst` closely. All systems prioritize SLO jobs by sacrificing BE jobs when load spikes. Cluster:SC256, Workload: E2E-LOAD-ℓ where $\ell \in [1.0,1.2,1.4,1.6]$ 50

4.11 3Sigma outperforms others on SLO Misses for a range of runtime variability. 3Sigma matches `PointPerfEst` in terms of SLO misses at the sacrifice of Best Effort goodput. Cluster:SC256. Workload:E2E-SAMPLE-n where $n \in [5, 10, 25, 50, 75, 100]$ 50
4.12 3Sigma scalability as a function of job submission per hour. Cluster: GOOGLE, Workload: SCALABILITY-n where $n \in [2000, 3000, 4000]$ 52

5.1 Compares the user provided CPU request amount to the actual average CPU usage of the jobs in the Google trace. (a) Only 20% of jobs are within 2X error range and 74% of jobs are over-estimated. (b) Further examination of over-estimates shows that a significant portion is significantly over-estimated. 55
5.2 Histogram of Precent Estimate Errors comparing estimates from the 3σPredict modified to provide estimates for CPU, Memory, CPU-time, Memory-time. Estimate Error values computed by $\frac{\text{estimate}-\text{actual}}{\text{actual}}$. Each datapoint is a bucket representing values within 5% of the nearest decile. The ”tail” datapoint inclues all estimate errors $> 95\%$. 56
5.3 Compares the estimate accuracy of jobs in first week and last week of the Google trace for Runtime, CPU, Memory, and CPU-time. Estimate accuracy is improved over time for all metrics, except for CPU-time. 57
5.4 Histogram of Percent Estimate Errors comparing runtime estimates computed from different predictors. ”Runtime” estimates are the runtime estimate from 3σPredict, while other estimates are computed by dividing CPU-time estimate from 3σPredict by the respective CPU values. 57
5.5 Compares the performance of `Dist` with other systems in the simulated cluster. `Dist` outperforms `Point` on SLO miss-rate and SLO Goodput while nearly matching `Point` and `Request` on BE Goodput and BE Latency. 69
5.6 Compares the amount of resource contention as `Dist` and other systems schedule jobs in the cluster. `Dist` outperforms `Point` on SLO miss-rate and SLO Goodput while nearly matching `Point` on BE Goodput and BE Latency. 70
5.7 Compares the performance of `Dist` with the systems with individual features disabled in the simulated cluster. 71
5.8 Compares the amount of resource contention as `Dist` with the systems with individual features disabled. 72
5.9 Compares the performance of `Dist` with other systems in the simulated cluster. `Dist` consistently outperforms `Point` on SLO miss-rate and SLO Goodput while nearly matching `Point` and `Request` on BE Goodput and BE Latency. . 75
5.10 Compares the amount of resource contention as `Dist` and other systems schedule jobs in the cluster. `Dist` outperforms `Point` on SLO miss-rate and SLO Goodput while nearly matching `Point` on BE Goodput and BE Latency. 76

List of Tables

2.1 Hardware characteristics of the clusters analyzed in this chapter. For the Google trace [72], (*) signifies a resource has been normalized to the largest node. 8

3.1 Summary of the characteristics of each trace. Note that the Google workload appears to be an outlier. 12

4.1 Scheduler approaches compared. 44
4.2 Absolute performance difference between real and simulation experiments. Workload:E2E. 47

Chapter 1

Introduction

Modern computing clusters support a mixture of diverse activities, ranging from customer-facing internet services, software development and test, scientific research, and exploratory data analytics [8, 72]. The role of the cluster schedulers is to map these tasks to the heterogeneous resources available in the cluster. They face a daunting task of efficiently matching the pending job according to their scheduling preferences (in terms of the resource and deadlines) while minimizing the completion latency and maximizing the cluster efficiency.

Many recent schedulers exploit knowledge of pending jobs' runtimes and resource usages as a powerful building block [24, 54, 88]. Using estimates of runtime and resource usage, a scheduler can pack jobs aggressively into its resource plan [24, 54, 88, 92], such as allowing a latency sensitive job to start before a high-priority batch job as long as the batch job will meet its deadline. The knowledge enables the scheduler to consider whether it is better to wait for a job's preferred resources to be freed or to start the job right away on sub-optimal resources [13, 88]. Knowledge of job runtime and resource usage leads to more robust scheduling decisions than using simple scheduling algorithms that cannot leverage this information.

In most cases, estimates come from the observation of similar jobs (e.g., from the same user or past instances from the same periodic job script) run in the past. A point runtime estimate (e.g., mean or median) is derived from the relevant subset of the history and used by the scheduler. If such estimates are accurate, schedulers relying on them outperform those using other approaches. Previous research [40, 88] suggests that these schedulers are robust to a reasonable degree of estimation error (e.g., up to 50%).

However, analyses of workloads from real clusters show that the actual estimate errors span much larger ranges than those previously explored. Analyses of user-provided resource requests in the Google cluster trace [72] show only 20% of the estimates are within a factor of two of the actual average resource usage, and a significant portion (74%) of jobs are over-estimated with the majority being off by more than an order of magnitude. Applying a state-of-the-art ML-based predictor [87] to three real-world traces shows good estimates in general (77%-92% are within a factor of two of the actual runtime and most much closer), but a significant percentage (8%-23%) of estimates are not within that range, and some are off by more than an order of magnitude (Chapters 4 and 5). Even very effective predictors suffer from inaccuracies and outliers because there is significant inherent variability in multi-purpose cluster workloads.

The impact of inaccurate point estimates on scheduler performance is significant. Testing with real estimate profiles reveals that a scheduler relying on such estimates performs much worse with real estimate error profiles as compared to having perfect estimates. The point-estimate based scheduler makes less informed decisions and struggles to handle difficult-to-estimate runtimes and resource usages.

The scheduler is often too optimistic and starts jobs with under-estimated runtimes later than it should, and it is often too conservative and neglects to schedule jobs that are predicted to not finish on time, even if the cluster resources are available. Effects of inaccurate resource usage estimates are more severe. The scheduler often packs too many tasks in the same machine, triggering resource contention in the machine, and it is often too hesitant in scheduling tasks resulting in low cluster utilization. Knowing only the point estimate, e.g. an average of a job runtime or resource usage, the scheduler cannot reason about the outcomes that may be significantly different from the average.

Instead, this dissertation proposes and evaluates systems that can leverage full distributions (e.g., the histogram of observed runtimes or resource usages) rather than single point estimates. A distribution provides much more information (e.g., variance, possible multi-modal behaviors, etc.) and enables the scheduler to make more robust decisions. By considering the range of possible runtimes or resource usages for a job, and their likelihoods, the scheduler can explicitly consider various potential outcomes from each possible scheduling option and select an option based on optimizing the expected outcome.

1.1 Thesis Statement

This dissertation explores the following thesis statement.

Schedulers that rely on information about job runtimes and resource usages can address imperfect predictions with up to 75% fewer deadline misses and 36% greater SLO goodput by looking at likelihoods of possible outcomes rather than single point expected outcomes.

The dissertation will provide the following evidence to support the thesis statement.

- **Diversity of cluster workloads [8]** (Chapter 3)
  This chapter presents an analysis of the private and HPC cluster traces that spans job characteristics, workload heterogeneity, resource utilization, and failure rates and contrast findings with the Google cluster trace characteristics. The analysis shows that the private cluster workloads, consisting of data analytics jobs expected to be more closely related to the Google workload, display more similarity to the HPC cluster workloads, suggesting that additional traces should be considered when evaluating the generality of new research. Characterization of real workloads also revealed that there exists inherent variability in the job runtimes and resource usage that cannot be captured by single point estimates. An evaluation of a history-based runtime predictor with four different traces demonstrates it is not trivial to obtain perfect runtime predictions in real workloads, especially if the predictor is provided with insufficient information.
- **3Sigma: a runtime distribution based scheduler [69]** (Chapter 4)
  Knowing how long each job will execute enables a scheduler to more effectively pack

jobs with diverse time concerns (e.g., deadline vs. the-sooner-the-better) and placement preferences on heterogeneous cluster resources. But, existing schedulers use single-point estimates, and this chapter shows that they are fragile in the face of real-world estimate error profiles. Instead of reducing relevant history to a single point, 3Sigma schedules jobs based on full distributions of relevant runtime histories and explicitly creates plans that mitigate the effects of anticipated runtime uncertainty. Experiments with workloads show that 3Sigma achieves 75% fewer deadline misses and 36% greater SLO goodput compared to a state-of-the-art scheduler that uses point estimates from a state-of-the-art predictor; in fact, the performance of 3Sigma approaches the end-to-end performance of a scheduler based on a hypothetical, perfect runtime predictor.

- **DistSched: a resource-runtime distribution based scheduler** (Chapter 5)
  An accurate knowledge of each job's resource usage benefits schedulers as the knowledge allows schedulers to safely pack jobs more tightly, increasing the cluster utilization while minimizing performance jitter due to resource contention. Most systems rely on user-provided estimates as a source of the knowledge, but an analysis of the Google cluster trace shows only a few jobs have an accurate estimate. Cluster administrators use various heuristics to tackle issues arising from the mis-estimates, but these are only a partial solution. This chapter describes DistSched, a resource-runtime distribution based scheduler and explores how lessons learned from 3Sigma apply to the problem of imperfect estimates arising from resource usage uncertainty. By leveraging distributions of resource usage from the relevant history, taking advantage of much richer information, and utilizing mitigation mechanisms, the resource-runtime distribution based scheduler can make robust decisions to mitigate the effects of resource uncertainty. Experiments with a subset of the Google cluster trace show that the resource-runtime distribution based scheduler achieves 49% fewer deadline misses and 5% greater SLO goodput compared to point-estimate based schedulers that depend on point estimates from a history-based predictor or user-provided resource requests and approaches the performance of a hypothetical system that uses a perfect runtime predictor.

1.2 Contributions

This dissertation makes the following contributions:

Diversity of cluster workloads [8]:

- It presents an analysis of the private and HPC cluster traces that spans job characteristics, workload heterogeneity, resource utilization, and failure rates and contrast findings with the Google cluster trace characteristics.
- It characterizes real workloads from three different environments revealing that there exists inherent variability in the job runtimes and resource usages that cannot be captured by single point estimates.
- It reports on an evaluation of a history-based runtime predictor with four different traces demonstrating that it is not trivial to obtain perfect runtime predictions in real workloads, especially if the predictor is provided with insufficient information.

3Sigma [69]:

- It describes a scheduler, called 3Sigma, that looks at runtime distributions instead of a point runtime estimate and can much more robustly address imperfect runtime predictions.
- It demonstrates that a runtime distribution of a job can be estimated from the history of jobs run in the past and shows that estimated distributions are effective for the workloads studied.
- It reports on large-scale experiments showing that 3Sigma is viable in practice, outperforms point-estimate based schedulers, and approaches the performance of a hypothetical system that has perfect knowledge of job runtimes.

DistSched:

- It describes a scheduler that looks at resource usage distributions instead of point estimates, as well as its mitigation mechanisms, demonstrating that such a scheduler can much more robustly address imperfect predictions.
- It demonstrates that a distribution of the resource usage of a job can be estimated from the relevant part of the job history and shows that estimated distributions are effective for the workloads studied.
- It reports on the results of experiments with the Google cluster trace demonstrating the efficacy of the distribution-based scheduling approach in coping with resource usage uncertainty.

1.3 Outline

The remainder of the dissertation is organized as follows. Chapter 2 motivates our work with more background on resource consolidation, cluster scheduling with job information, predicting job information, and the workloads discussed in the dissertation. Chapter 3 describes the workload analysis [8] comparing the Google trace with the private and HPC cluster traces. Chapter 4 describes 3Sigma [69], my scheduler that leverages distributions of the relevant runtime histories rather than just a point estimate derived from it. Chapter 5 describes DistSched, my scheduler that can leverage distribution of the resource usage (cpu, memory, and cpu-time) and account the risk of contention to make robust scheduling decisions. Chapter 6 concludes the dissertation and discusses future research directions.

Chapter 2

Background

Cluster schedulers are typically a component of the cluster orchestration system (e.g. YARN [89], Kubernetes [14], etc.) that manages the lifecycle of the cluster resources and jobs running in the system. In this model, users submit job specifications consisting of one or more tasks to the cluster manager, often times with the resource requirements (e.g. how much cpu and memory is needed and how long will it use these resources). The scheduler decides when and which machine to run each task of the job. Each task will execute within a container for resource isolation and security.

Cluster scheduling enjoys a long history of research, but increasing cluster consolidation and the emergence of a diverse mix of workload types stimulates a continuous stream of new innovations. This chapter describes the additional background and research related to the dissertation.

2.1 Resource consolidation

Increasing amount of applications are now hosted on data-centers, both in public clouds [4, 5, 6] and private in-house data-centers with frameworks [3, 14, 46, 89]. By consolidating different types of workloads to the same shared data-centers, cluster administrators expect a lower total cost of ownership through economies of scale. It also offers flexibility for the users as they can leverage the same environment to launch different types of jobs ranging from batch analytics to long running services.

However, resource consolidation also presents a set of new challenges to cluster schedulers.

1) To support diverse types of workloads, data-centers are increasingly becoming heterogeneous. Even if a data-center is newly constructed, it may consist of machines with different hardware configurations. The variety will only increase over time as new types of machines are introduced to the clusters [12, 72]. Hardware accelerators such as FGPAs [1, 2] or TPUs [53] are now commonplace. Schedulers need to efficiently map pending work to the heterogeneous resources so as to satisfy their diverse scheduling concerns.

2) Low utilization is a major challenge for cloud facilities [17, 28, 60, 61, 72], even for clusters that encourage sharing of the resources across different workloads. This is mainly due to a disparity between user resource requests and actual resource usage, which recent research efforts try to alleviate through workload characterization and aggressive consolidation [28, 57, 58].

3) Aggressive packing to achieve high utilization does not work well with latency-critical

services due to interference [57]. To ensure minimal interference, applications are typically profiled and classified according to historical data [28, 57].

As a result, existing scheduling systems for traditional compute clusters fail to perform well [88, 92] and rely on more sophisticated scheduling algorithms.

2.2 Cluster scheduling with job information

Modern schedulers use the knowledge of pending jobs' runtime and resource usage as a powerful building block. Accurate job runtime and resource usage information can be exploited to significant benefit in at least four ways at schedule-time.

1) Cluster workloads are increasingly a mixture of business-critical production jobs and best-effort engineering/analysis jobs. The production jobs, often submitted by automated systems [50, 83], tend to be resource-heavy and to have strict completion deadlines [24, 54]. The best-effort jobs, such as exploratory data analytics and software development/debugging, while lower priority, are often latency-sensitive. Given runtime estimates, schedulers can more effectively pack jobs, simultaneously meets more deadlines for production jobs and reducing average latency for best-effort jobs [24, 54, 88].

2) Datacenter resources are increasingly heterogeneous, and some jobs behave differently (e.g., complete faster) depending upon which machine(s) they are assigned to. Maximizing cluster effectiveness in the presence of jobs with such considerations can be more effective when job runtimes are known [13, 88, 101].

3) Many parallel computations can only run when all tasks comprising them are initiated and executed simultaneously (gang-scheduling) [65, 68]. Maximizing resource utilization while arranging for such bulk resource assignments is easier when job runtimes are known.

4) Resource under-utilization can be alleviated through workload characterization and aggressive consolidation [28, 57, 58]. Given accurate knowledge of job resource usage, schedulers can control performance variation due to resource contention to meet jobs' performance service-level objectives [28, 33].

2.3 Predicting job information

Most systems [14, 46, 89] now expect users to provide resource requirements when a job is submitted to the system. In some environments, especially HPC and grid computing environments, users are expected to provide runtime information explicitly. Naturally, the quality of such user-provided information varies widely, and automated approaches to generating predictions is desirable. Different strategies for the prediction can be used based on the amount of assumption or knowledge about the workload.

Some techniques [24, 25, 52, 54, 56, 91] are designed for explicitly repeating jobs, such as in a scripted simulation parameter sweep or regular post-processing of an output file. In this scenario, each such job is a recurrence of a nearly identical job with known historical information.

Performance modeling based *white-box techniques* can be used if the structure of each job is known in advance. For example, Jockey [33] and Perforator [70] leverage job structure and combine it with profiling for accurate predictions. MapReduce's map-shuffle-reduce structure is well-understood and lends itself to analytical performance models, such as ARIA [94] and Parallax [64]. Similarly, Apollo [13], Ernest [90], SLAOrchestrator [67], and Islam et al. [51] rely on leveraging job structure knowledge to estimate job runtimes and resource usage.

Lastly, some predictors use *black-box techinques* to address jobs that do not arrive with explicit recurrence nor performance models. In the absence of any other information, Harchol-Balter and Downey [43] or Kairos [27] assumes the job is half-way completed. Other predictors [23, 81, 87] assume, even in multi-purpose clusters for a diverse array of activities, most jobs will be similar to some subset of previous jobs. These systems identify and determine an estimate (e.g., mean or median) from the relevant part of the history. The assumptions made by the systems discussed in this dissertation (Chapters 4 and 5) fall in this category.

2.4 Workloads used in this dissertation

Despite the intense activity in the areas of cloud and job scheduling research, publicly available cluster workload datasets remain scarce. The three major dataset sources today are: the Google cluster trace [72] collected in 2011, the Parallel Workload Archive [32] of High Performance Computing (HPC) traces collected since 1993, and the SWIM traces released in 2011 [21]. Of these, the Google trace has been used in more than 450 publications making it the most popular trace by far.

This dissertation introduces four new traces: two from the private cloud of Two Sigma, a hedge fund, and two from HPC clusters located at the Los Alamos National Laboratory (LANL)[1]. The Two Sigma traces are the longest, non-academic private cluster traces to date, spanning 9 months and more than 3 million jobs. The two HPC traces I introduce are also unique. The first trace, LANL Mustang, spans the entire 5-year lifetime of a general-purpose HPC cluster, making it the longest public trace to date, while also exhibiting shorter jobs than existing public HPC traces. The second trace, LANL OpenTrinity, originates from the 300,000-core current flagship supercomputer at LANL, making it the largest cluster with a public trace, to my knowledge.

I evaluate the systems in the dissertation using *some* of the workloads as appropriate. Specifically, 3Sigma is evaluated using the Google, Two Sigma, and Mustang cluster traces. The OpenTrinity cluster trace is not used for evaluation as it has demonstrated low predictability (Sec. 3.6.2), potentially caused by the shorter duration of the trace or inconsistency in the workload during the OpenScience configuration period. DistSched is only evaluated using the Google cluster workload, because the other traces do not contain resource utilization information that is crucial for simulating the workload for experiments.

The hardware configuration of each cluster is shown in Table 2.1. Rest of this section discusses each cluster is in more detail.

2.4.1 Google cluster

In 2012, Google released a trace of jobs that ran in one of their compute clusters [72]. It is a 29-day trace consisting of 672074 jobs and 48 million tasks, some of which were issued

---

[1] The LANL traces were released and are available to the public at `http://www.pdl.cmu.edu/ATLAS`

Platform	Nodes	CPUs	RAM	Length
LANL OpenTrinity	9408	32	128GB	3 months
LANL Mustang	1600	24	64GB	5 years
TwoSigma A	872	24	256GB	9 months
TwoSigma B	441	24	256GB	
Google B	6732	0.50*	0.50*	29 days
Google B	3863	0.50*	0.25*	
Google B	1001	0.50*	0.75*	
Google C	795	1.00*	1.00*	
Google A	126	0.25*	0.25*	
Google B	52	0.50*	0.12*	
Google B	5	0.50*	0.03*	
Google B	5	0.50*	0.97*	
Google C	3	1.00*	0.50*	
Google B	1	0.50*	0.06*	

Table 2.1: Hardware characteristics of the clusters analyzed in this chapter. For the Google trace [72], (*) signifies a resource has been normalized to the largest node.

through the MapReduce framework, and ran on 12583 heterogeneous nodes in May 2011. The workload consists of both long-running services and batch jobs [95]. Google has not released the exact hardware specifications of each cluster node. Instead, as shown in Table 2.1, nodes are presented through anonymized platform names representing machines with different combinations of microarchitectures and chipsets [98]. Note that the number of CPU cores and RAM for each node in the trace have been normalized to the most powerful node in the cluster. In the analysis, I estimate the total number of cores in the Google cluster to be 106544. I derive this number by assuming that the most popular node type (Google B with 0.5 CPU cores) is a dual-socket server, carrying quad-core AMD Opteron Barchelona CPUs that Google allegedly used in their data-centers at the time [44]. Unlike previous workloads, jobs can be allocated fractions of a CPU core [78].

2.4.2 Two Sigma clusters

The private workload traces I introduce originate from two datacenters of Two Sigma, a hedge fund firm. The workload consists of data analytics jobs processing financial data. A fraction of these jobs are handled by a Spark [84] installation, while the rest are serviced by home-grown data analytics frameworks. The dataset spans 9 months of the two data-centers' operation starting in January 2016, covering a total of 1313 identical compute nodes with 31512 CPU cores and 328TB RAM. The logs contain 3.2 million jobs and 78.5 million tasks, collected by an internally-developed job scheduler running on top of Mesos [46]. Because both datacenters experience the same workload and consist of homogeneous nodes, I collectively refer to both data sources as the *TwoSigma* trace and analyze them together.

2.4.3 LANL Mustang cluster

Mustang was an HPC cluster used for *capacity computing* at LANL from 2011 to 2016. Capacity clusters such as Mustang are architected as cost-effective, general-purpose resources for a large number of users. Mustang was largely used by scientists, engineers, and software developers at LANL and it was allocated to these users at the granularity of physical nodes. The cluster consisted of 1600 identical compute nodes, with a total of 38400 AMD Opteron 6176 2.3GHz cores and 102TB RAM.

The Mustang dataset covers the entire 61 months of the machine's operation from October 2011 to November 2016, which makes this the longest publicly available cluster trace to date. The Mustang trace is also unique because its jobs are shorter than those in existing HPC traces. Overall, it consists of 2.1 million multi-node jobs submitted by 565 users and collected by SLURM [76], an open-source cluster resource manager. The fields available in the trace are similar to those in the TwoSigma trace, with the addition of a time budget field per job, that if exceeded causes the job to be killed.

2.4.4 LANL OpenTrinity supercomputer

In 2018, OpenTrinity is the largest supercomputer at LANL and it is used for *capability computing*. Capability clusters are a large-scale, high-demand resource introducing novel hardware technologies that aid in achieving crucial computing milestones, such as higher-resolution climate and astrophysics models. OpenTrinity's hardware was stood up in two pre-production phases before being put into full production use and the trace was collected before the second phase completed. At the time of data collection, OpenTrinity consisted of 9408 identical compute nodes, a total of 301056 Intel Xeon E5-2698v3 2.3GHz cores and 1.2PB RAM, making this the largest cluster with a publicly available trace by number of CPU cores.

The OpenTrinity dataset covers 3 months from February to April 2017. During that time, OpenTrinity was operating in OpenScience mode, i.e., the machine was undergoing beta testing and was available to a wider number of users than after it receives its final security classification. I note that OpenScience workloads are representative of a capability supercomputer's workload, as they occur roughly every 18 months when a new machine is introduced, or before an older one is decommissioned. The dataset, which I will henceforth refer to as *OpenTrinity*, consists of 25237 multi-node jobs issued by 88 users and collected by MOAB [7], an open-source cluster scheduling system. The information available in the trace is the same as that in the Mustang trace.

Chapter 3

Diversity of cluster workloads

Despite intense activity in the areas of cloud and job scheduling research, publicly available cluster workload datasets remain scarce. The three major dataset sources today are: the Google cluster trace [72] collected in 2011, the Parallel Workload Archive [32] of High Performance Computing (HPC) traces collected since 1993, and the SWIM traces released in 2011 [21]. Of these, the Google trace has been used in more than 450 publications making it the most popular trace by far. Unfortunately, this 29-day trace is often the only one used to evaluate new research. By contrasting its characteristics with newer traces from different environments, I have found that the Google trace alone is insufficient to accurately prove the generality of a new technique.

The goal is to uncover overfitting of prior work to the characteristics of the Google trace. To achieve this, my first contribution is an analysis examining the generality of workload characteristics derived from the Google trace, when four new traces are considered. Overall, I find that the private Two Sigma cluster workloads display similar characteristics to HPC, despite consisting of data analytics jobs that more closely resemble the Google workload. Table 3.1 summarizes all my findings. For those characteristics where the Google workload is an outlier, I have surveyed the literature and list affected prior work. In total, I surveyed 450 papers that reference the Google trace study [72] to identify popular workload assumptions, and I constrast them to the Two Sigma and LANL workloads to detect violations. I group the findings into four categories: job characteristics (Section 3.2), workload heterogeneity (Section 3.3), resource utilization (Section 3.4), and failure analysis (Section 3.5).

The findings suggest that evaluating new research using the Google trace alone is insufficient to guarantee generality. I further present a case study on the importance of dataset plurality and diversity when evaluating new research. For demonstration I use JVuPredict, the job runtime predictor of the JamaisVu scheduling system [87]. Originally, JVuPredict was evaluated using only the Google trace [87]. Evaluating its performance with the four new traces, however, helped us identify features that make it easier to detect related and recurring jobs with predictable behavior. This enabled us to quantify the importance of individual trace fields in runtime prediction. I describe the findings in Section 3.6.

Finally, I briefly discuss the importance of trace length in accurately representing a cluster's workload in Section 3.7. I list related work studying cluster traces in Section 3.8, before concluding.

Section	Characteristic	Google	TwoSigma	Mustang	OpenTrinity
Job Characteristics (§3.2)	Majority of jobs are small	✓	✗	✗	✗
	Majority of jobs are short	✓	✗	✗	✗
Workload Heterogeneity (§3.3)	Diurnal patterns in job submissions	✗	✓	✓	✓
	High job submission rate	✓	✓	✗	✗
Resource Utilization (§3.4)	Resource over-commitment	✓	✗	✗	✗
	Sub-second job inter-arrival periods	✓	✓	✓	✓
	User request variability	✗	✓	✓	✓
Failure Analysis (§3.5)	High fraction of unsuccessful job outcomes	✓	✓	✗	✓
	Jobs with unsuccessful outcomes consume significant fraction of resources	✓	✓	✗	✗
	Longer/larger jobs often terminate unsuccessfully	✓	✗	✗	✗

Table 3.1: Summary of the characteristics of each trace. Note that the Google workload appears to be an outlier.

3.1 Dataset information

As discussed in Sec. 2.4, I introduce four sets of job scheduler logs that were collected from a general-purpose cluster and a cutting-edge supercomputer at LANL, and across two clusters of Two Sigma, a hedge fund.

Users typically interact with the cluster scheduler by submitting commands that spawn multiple processes, or *tasks*, distributed across cluster nodes to perform a specific computation. Each such command is considered to be a *job* and users often compose scripts that generate more complex, multi-job schedules. In HPC clusters, where resources are allocated at the granularity of physical nodes similar to Emulab [15, 29, 45, 97], tasks from different jobs are never scheduled on the same node. This is not necessarily true in private clusters like Two Sigma.

3.2 Job characteristics

Many instances of prior work in the literature rely on the assumption of heavy-tailed distributions to describe the size and duration of individual jobs [9, 19, 25, 26, 71, 85]. In the LANL and TwoSigma workloads these tails appear significantly lighter.

Observation 1: *On average, jobs in the TwoSigma and LANL traces request 3 - 406 times more CPU cores than jobs in the Google trace. Job sizes in the LANL traces are more uniformly distributed.*

Figure 3.1 shows the Cumulative Distribution Functions (CDFs) of job requests for CPU cores across all traces, with the x-axis in logarithmic scale. I find that the 90% of smallest jobs in the Google trace request 16 CPU cores or fewer. The same fraction of TwoSigma jobs request 108 cores, and 1-16K cores in the LANL traces. Very large jobs are also more common outside Google. This is unsurprising for the LANL HPC clusters, where allocating thousands of CPU

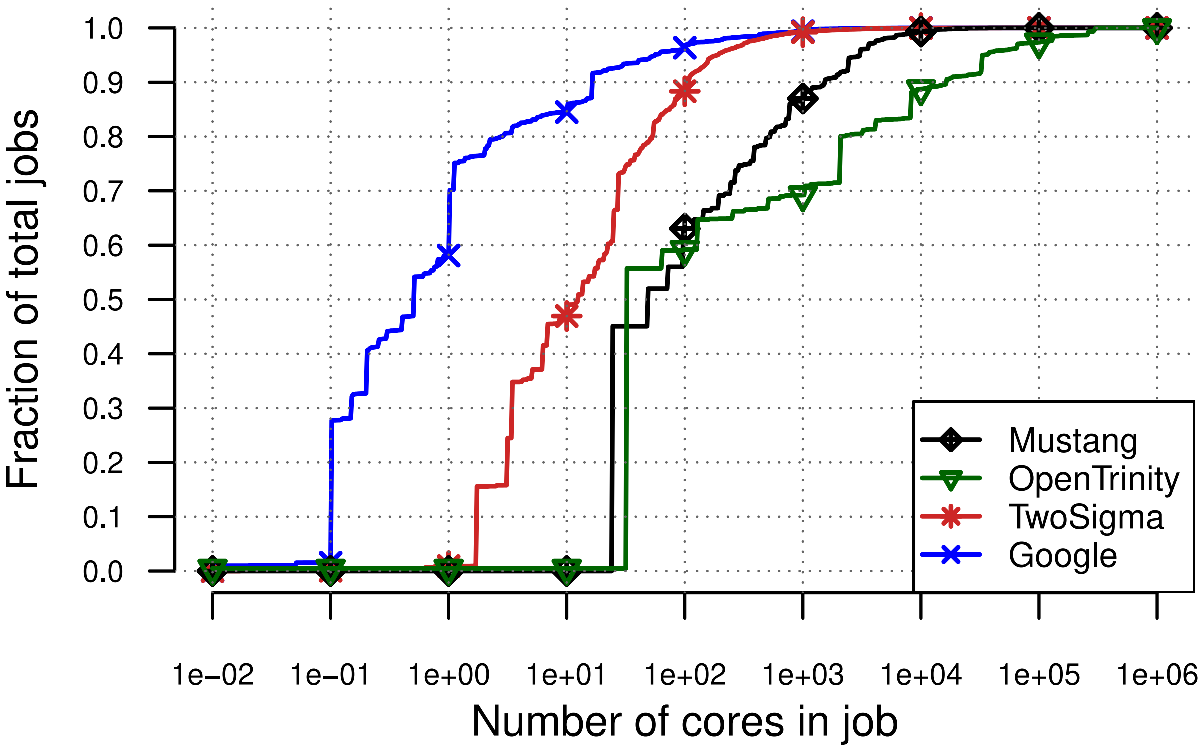


Figure 3.1: CDF of job sizes based on allocated CPU cores.

cores to a single job is not uncommon, as the clusters' primary use is to run massively parallel scientific applications. It is interesting to note, however, that while the TwoSigma clusters contain fewer cores than the other clusters I examine (3 times fewer than the Google cluster), its median job is more than an order of magnitude larger than a job in the Google trace. An analysis of allocated memory yields similar trends.

Observation 2: *The median job in the Google trace is 4-5 times shorter than in the LANL or TwoSigma traces. The longest 1% of jobs in the Google trace, however, are 2-6 times longer than the same fraction of jobs in the LANL and TwoSigma traces.*

Figure 3.2 shows the CDFs of job durations for all traces. I find that in the Google trace, 80% of jobs last less than 12 minutes *each*. In the LANL and TwoSigma traces jobs are at least an order of magnitude longer. In TwoSigma, the same fraction of jobs last up to 2 hours and in LANL, they last up to 3 hours for Mustang and 6 hours for OpenTrinity. Surprisingly, the tail end of the distribution is slightly shorter for the LANL clusters than for the Google and TwoSigma clusters. The longest job is 16 hours on Mustang, 32 hours in OpenTrinity, 200 hours in TwoSigma, and at least 29 days in Google (the duration of the trace). For LANL, this is due to hard limits causing jobs to be indiscriminately killed. For Google, the distribution's long tail is likely attributed to long-running services.

Implications. These observations impact the immediate applicability of job scheduling approaches whose efficiency relies on the assumption that the vast majority of jobs' durations are in the order of minutes, and job sizes are insignificant compared to the size of the cluster. For example, Ananthanarayanan et al. [9] propose to mitigate the effect of stragglers by duplicating

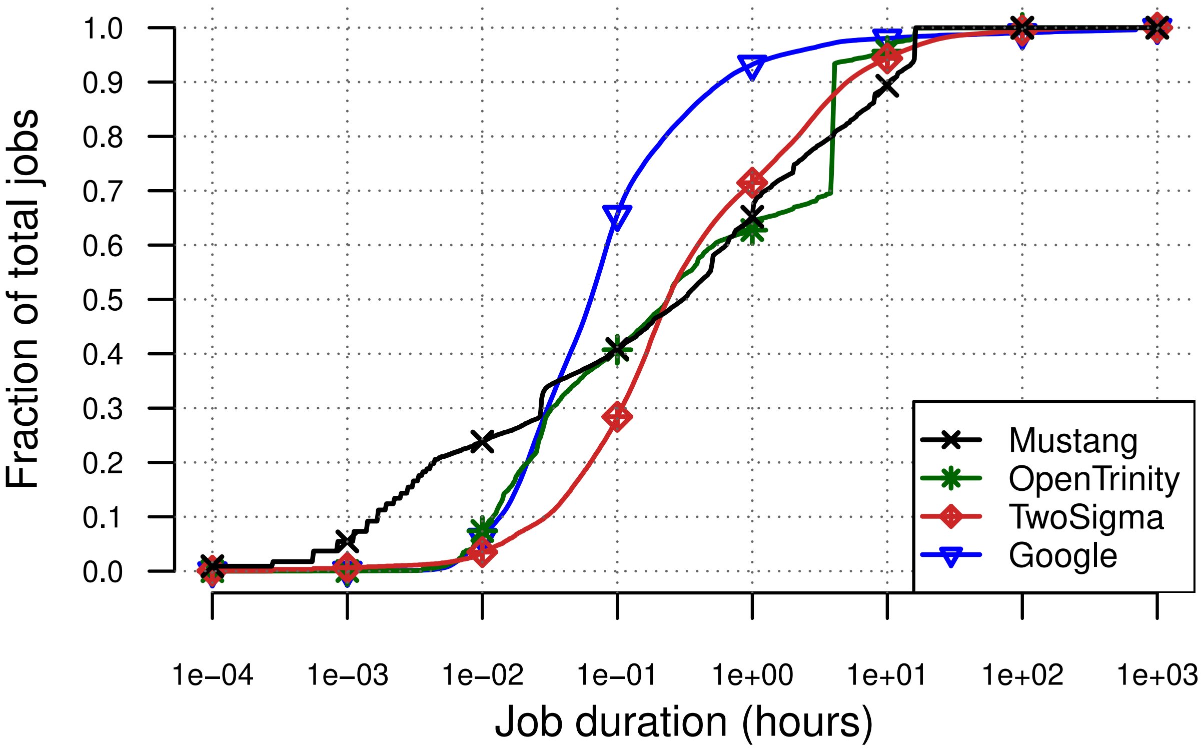


Figure 3.2: CDF of the durations of individual jobs.

tasks of smaller jobs. This is an effective approach for Internet service workloads (Microsoft and Facebook are represented in the paper) because the vast majority of jobs can benefit from it, without significantly increasing the overall cluster utilization. For the Google trace, for example, 90% of jobs request less than 0.01% of the cluster each, so duplicating them only slightly increases cluster utilization. At the same time, 25-55% of jobs in the LANL and TwoSigma traces *each* request *more than* 0.1% of the cluster's cores, decreasing the efficiency of the approach and suggesting replication should be used judiciously. This does not consider that LANL tasks are also tightly-coupled and the entire job has to be duplicated.

Another example is the work by Delgado et al. [25], which improves the efficiency of distributed schedulers for short jobs by dedicating them a fraction of the cluster. This partition ranges from 2% for Yahoo and Facebook traces, to 17% for the Google trace where jobs are significantly longer, to avoid increasing job service times. For the TwoSigma and LANL traces I have shown that jobs are even longer than for the Google trace (Figure 3.2), so larger partitions will likely be necessary to achieve similar efficiency. At the same time, jobs running in the TwoSigma and LANL clusters are also larger (Figure 3.1), so service times for long jobs are expected to increase unless the partition is shrunk. Other examples of work that is likely affected include task migration of short and small jobs [85] and hybrid scheduling aimed on improving head-of-line blocking for short jobs [26].

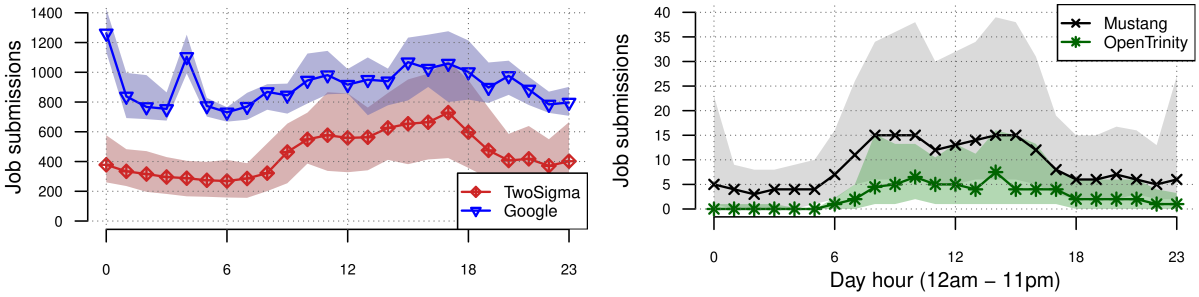


Figure 3.3: Hourly job submission rates for a given day. The lines represent the median, while the shaded region shows the distance between the 25^{th} and 75^{th} percentiles.

3.3 Workload heterogeneity

Another common assumption about cloud workloads is that they are characterized by heterogeneity in terms of resources available to jobs, and job interarrival times [18, 37, 49, 78, 96]. The private and HPC clusters I study, however, consist of homogeneous hardware (see Table 2.1) and user activity follows well-defined diurnal patterns, even though the rate of scheduling requests varies significantly across clusters.

Observation 3: *Diurnal patterns are universal. Clusters received more scheduling requests and smaller jobs at daytime, with minor deviations for the Google trace.*

In Figure 3.3 I show the number of job scheduling requests for every hour of the day. I choose to show metrics for the median day surrounded by the other two quartiles because the high variation across days causes the averages to be unrepresentative of the majority of days (see Section 3.7). Overall, diurnal patterns are evident in every trace and user activity is concentrated at daytime (7AM to 7PM), similar to prior work [63]. An exception to this is the Google trace, which is most active from midnight to 4AM, presumably due to batch jobs leveraging the available resources.

Sizes of submitted jobs are also correlated with the time of day. I find that longer, larger jobs in the LANL traces are typically scheduled during the night, while shorter, smaller jobs tend to be scheduled during the day. The reverse is true for the Google trace, which prompts my earlier assumption on nightly batch jobs. Long, large jobs are also scheduled at daytime in the TwoSigma clusters, despite having a diurnal pattern similar to LANL clusters. This is likely due to TwoSigma's workload consisting of financial data analysis, which bears a dependence on stock market hours.

Observation 4: *Scheduling request rates differ by up to 3 orders of magnitude across clusters. Sub-second scheduling decisions seem necessary in order to keep up with the workload.*

One more thing to take away from Figure 3.3 is that the rate of scheduling requests can differ significantly across clusters. For the Google and TwoSigma traces, hundreds to thousands of jobs are submitted every hour. On the other hand, LANL schedulers never receive more than 40 requests on any given hour. This could be related to the workload or the number of users in the system, as the Google cluster serves 2 times as many user IDs as the Mustang cluster and 9 times as many as OpenTrinity.

Implications: Previous work such as Omega [78] and ClusterFQ [96] propose distributed

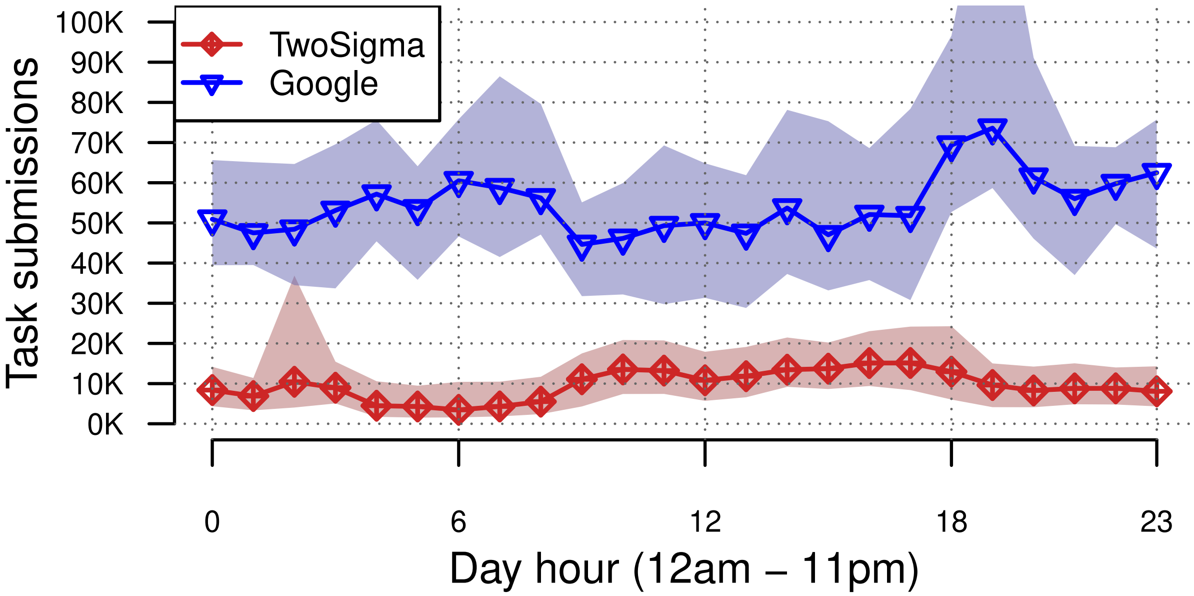


Figure 3.4: Hourly task placement requests for a given day. The lines represent the median, while the shaded region shows the distance between the 25^{th} and 75^{th} percentiles.

scheduling designs especially applicable to heterogeneous clusters. This does not seem to be an issue for environments such as LANL and TwoSigma, which intentionally architect homogeneous clusters to lower performance optimization and administration costs.

As cluster sizes increase, so does the rate of scheduling requests, urging us to reexamine prior work. Quincy [49] represents scheduling as a Min-Cost Max-Flow (MCMF) optimization problem over a task-node graph and continuously refines task placement. The complexity of this approach, however, becomes a drawback for large-scale clusters such as the ones I study. Gog et al. [37] find that Quincy requires 66 seconds (on average) to converge to a placement decision in a 10,000-node cluster. The Google and LANL clusters I study already operate on that scale (Table 2.1). I have shown in Figure 3.3 that the average frequency of job submissions in the LANL traces is one job every 90 seconds, which implies that this scheduling latency may work, but this will not be the case for long. OpenTrinity is currently operating with 19,000 nodes and, under the DoE's Exascale Computing Project [66], 25 times larger machines are planned within the next 5 years. Note that when discussing scheduling so far I refer to *jobs*, since HPC jobs have a gang scheduling requirement. Placement algorithms such as Quincy, however, focus on *task* placement.

An improvement to Quincy is Firmament [37], a centralized scheduler employing a generalized approach based on a combination of MCMF optimization techniques to achieve sub-second task placement latency on average. As Figure 3.4 shows, sub-second latency is paramount, since the rate of task placement requests in the Google and TwoSigma traces can be as high as 100K requests per hour, i.e. one task every 36ms. Firmament's placement latency, however, increases to several seconds as cluster utilization increases. For the TwoSigma and Google traces this can be problematic.

3.4 Resource utilization

A well-known motivation for the cloud has been resource consolidation, with the intention of reducing equipment ownership costs. An equally well-known property of the cloud, however, is that its resources remain underutilized [17, 28, 60, 61, 72]. This is mainly due to a disparity between user resource requests and actual resource usage, which recent research efforts try to alleviate through workload characterization and aggressive consolidation [28, 57, 58]. The analysis finds that user resource requests in the LANL and TwoSigma traces are characterized by higher variability than in the Google trace. I also look into job inter-arrival times and how they are approximated when evaluating new research.

Observation 5: *Unlike the Google cluster, none of the other clusters I examine overcommit resources.*

Overall, I find that the fraction of CPU cores allocated to jobs is stable over time across all the clusters I study. For Google, CPU cores are over provisioned by 10%, while for other clusters unallocated cores range between 2-12%, even though resource overprovisioning is supported by their schedulers. Memory allocation numbers follow a similar trend. Unfortunately, the LANL and TwoSigma traces do not contain information on actual resource utilization. As a result, I can neither confirm, nor contradict results from earlier studies on the imbalance between resource allocation and utilization. What differs between organizations is the motivation for keeping resources utilized or available. For Google [72], Facebook [21], and Twitter [28], there is a tension between the financial incentive of maintaining only the necessary hardware to keep operational costs low and the need to provision for peak demand, which leads to low overall utilization. For LANL, clusters are designed to accommodate a predefined set of applications for a predetermined time period and high utilization is planned as part of efficiently utilizing federal funding. For the TwoSigma clusters, provisioning for peak demand is more important, even if it leads to low overall utilization, since business revenue is heavily tied to the response times of their analytics jobs.

Observation 6: *The majority of job interarrivals periods are sub-second in length.*

Interarrival periods are a crucial parameter of an experimental setup, as they dictate the load on the system under test. Two common configurations are second-granularity [28] or Poisson-distributed interarrivals [47], and I find that neither characterizes interarrivals accurately. In Figure 3.5 I show the CDFs for job interarrival period lengths. I observe that 44-62% of interarrival periods are sub-second, implying that jobs arrive at a faster rate than previously assumed. Furthermore, my attempts to fit a Poisson distribution on this data have been unsuccessful, as Kolmogorov-Smirnov tests [62] reject the null hypothesis with p-values $< 2.2 \times 10^{-16}$. This result does not account for a scenario where there is an underlying Poisson process with a rate parameter changing over time, but it suggests that caution should be used when a Poisson distribution is assumed.

Another common assumption is that jobs are very rarely big, i.e., made up of multiple tasks [47, 96]. In Figure 3.6 I show the CDFs for the number of tasks per job across organizations. I observe that 77% of Google jobs are single-task jobs, but the rest of the clusters carry many more multi-task jobs. I note that the TwoSigma distribution approaches that of Google only for

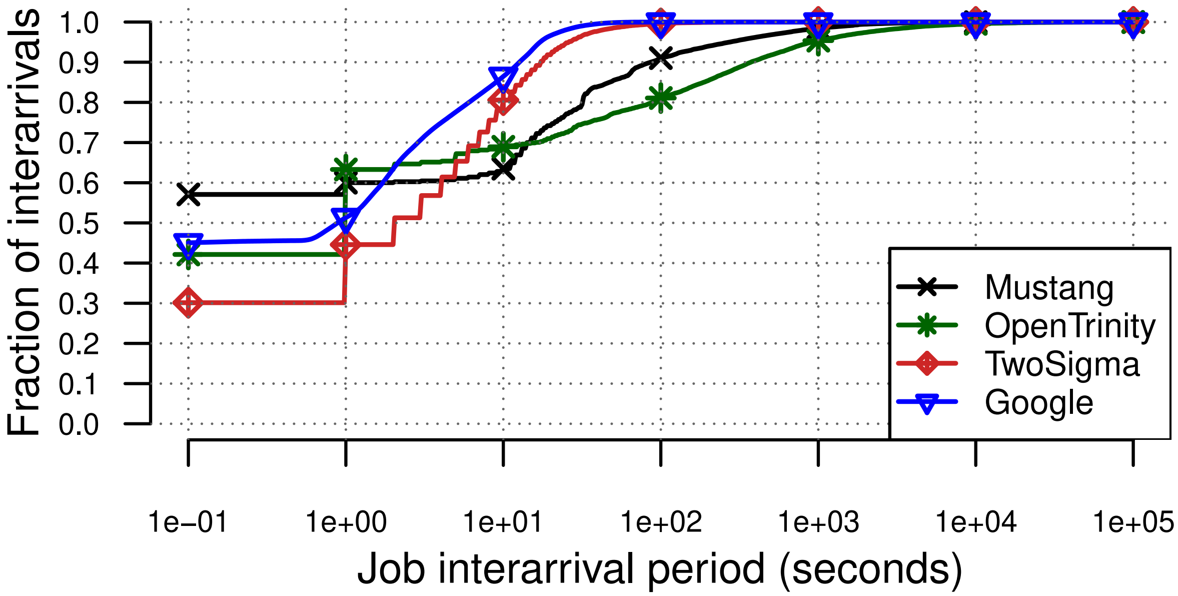


Figure 3.5: CDF of job interarrival times.

larger jobs. This suggests that task placement may be a harder problem outside Google, where single-task jobs are common, exacerbating the evaluation issues I outlined in Section 3.3 for existing task placement algorithms.

Observation 7: *User resource requests are more variable in the LANL and TwoSigma traces than in the Google trace.*

Resource under-utilization can be alleviated through workload consolidation. To ensure minimal interference, applications are typically profiled and classified according to historical data [28, 57]. The analysis suggests that this approach is likely to be less successful outside the Internet services world. To quantify variability in user behavior I examine the Coefficient of Variation[1] (CoV) across all requests of individual users. For the Google trace I find that the majority of users issue jobs within 2x of their average request in CPU cores. For the LANL and TwoSigma traces, on the other hand, 60-80% of users can deviate by 2-10x of their average request.

Implications: A number of earlier studies of Google [72], Twitter [28], and Facebook [21] data have highlighted the imbalance between resource allocation and utilization. Google tackles this issue by over-committing resources, but this is not the case for LANL and TwoSigma. Another proposed solution is Quasar [28], a system that consolidates workloads while guaranteeing a predefined level of QoS. This is achieved by profiling jobs at submission time and classifying them as one of the previously encountered workloads; misclassifications are detected by inserting probes in the running application. For LANL, this approach would be infeasible. First, jobs cannot be scaled down for profiling, as submitted codes are often carefully configured for the requested allocation size. Second, submitted codes are too complex to be accurately profiled in seconds, and probing them at runtime to detect misclassifications can introduce performance jitter that is prohibitive in tightly-coupled HPC applications. Third, in the LANL traces I often

---

[1] The Coefficient of Variation is a unit-less measure of spread, derived by dividing a sample's standard deviation by its mean.

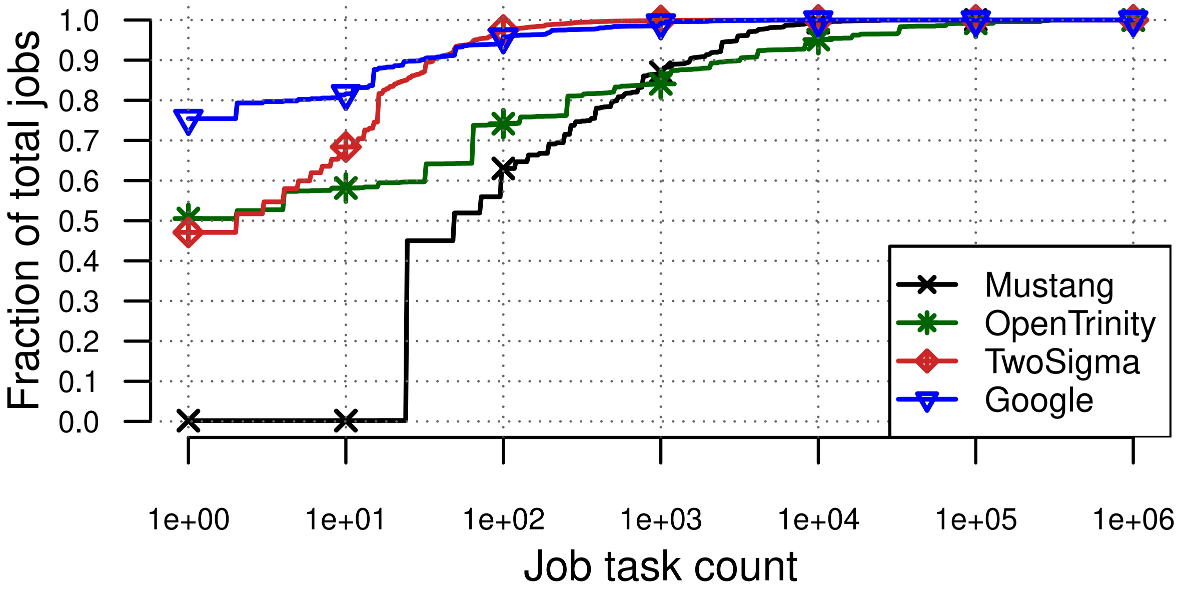


Figure 3.6: CDF of the number of tasks per job.

find that users tweak jobs before resubmitting them, as they re-calibrate simulation parameters to achieve a successful run, which is likely to affect classification accuracy. Fourth, resources are carefully reserved for workloads and utilization is high, which makes it hard to provision resources for profiling. For the TwoSigma and Google traces Quasar may be a better fit, however, at the rate of 2.7 jobs per second (Figure 3.3), 15 seconds of profiling [28] at submission time would result in an expected load of 6 jobs being profiled together. Since Quasar requires 4 parallel and isolated runs to collect sufficient profiling data, we would need resources to run at least 360 VMs concurrently, with guaranteed performance isolation between tham to keep up with the average load. This further assumes the profiling time does not need to be increased beyond 15 seconds. Finally, Quasar [28] was evaluated using multi-second inter-arrival periods, so testing would be necessary to ensure that one order of magnitude more load can be handled (Figure 3.5), and that it will not increase the profiling cost further.

Another related approach to workload consolidation is provided by TSF [96], a scheduling algorithm that attempts to maximize the number of task slots allocated to each job, without favoring bigger jobs. This ensures that the algorithm remains starvation-free, however it results in significant slowdowns in the runtime of jobs with 100+ tasks, which the authors define as big. This would be prohibitive for LANL, where jobs must be scheduled as a whole, and such "big" jobs are much more prevalent and longer in duration. Other approaches for scheduling and placement assume the availability of resources that may be unavailable in the clusters I study here, and their performance is shown to be reduced in highly-utilized clusters [41, 47].

3.5 Failure analysis

Job scheduler logs are often analyzed to gain an understanding of job failure characteristics in different environments [20, 30, 34, 35, 74]. This knowledge allows for building more robust

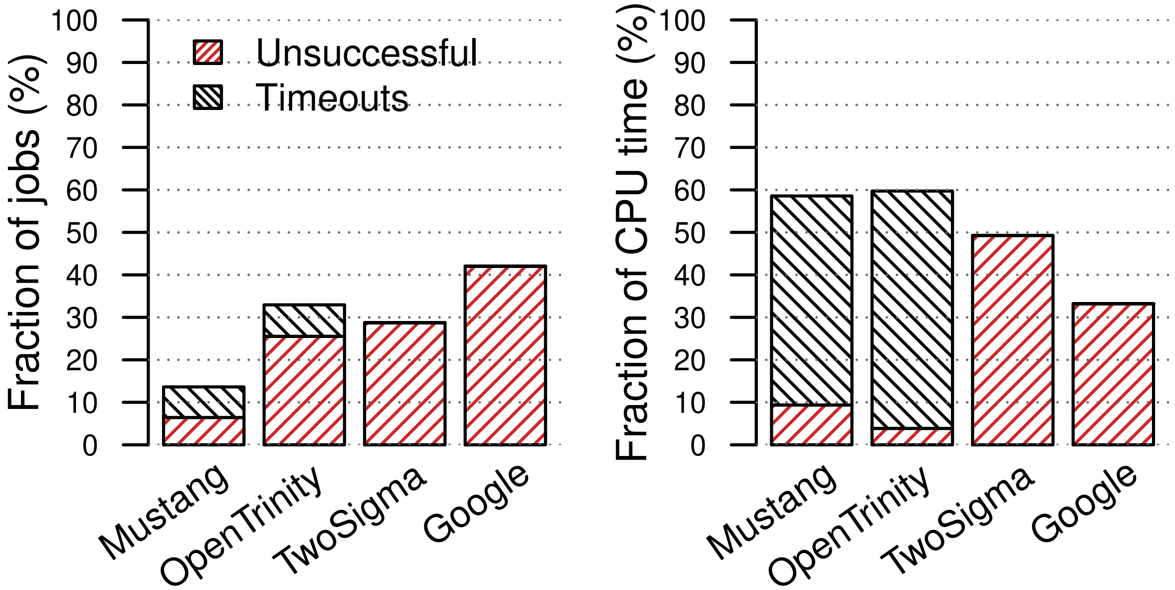


Figure 3.7: Breakdown of the total number of jobs, as well as CPU time, by job outcome.

systems, which is especially important as we transition to exascale computing systems where failures are expected every few minutes [82], and cloud computing environments built on complex software stacks that increase failure rates [20, 75].

Definitions. An important starting point for any failure analysis is defining what constitutes a failure event. Across all traces I consider, I define as *failed jobs* all those that end due to events whose occurrence was not intended by users or system administrators. I do not distinguish failed jobs by their root cause, e.g., software and hardware issues, because this information is not reliably available. There are other job termination states in the traces, in addition to success and failure. For the Google trace, jobs can be killed by users, tasks can be evicted in order to schedule higher-priority ones, or have an unknown exit status. For the LANL traces, jobs can be cancelled intentionally. I group all these job outcomes as *aborted jobs* and collectively refer to failed and aborted jobs as ***unsuccessful jobs***.

There is another job outcome category. At LANL, users are required to specify a runtime estimate for each job. This estimate is treated as a time limit, similar to an SLO, and the scheduler kills the job if the limit is exceeded. I refer to these killings as ***timeout jobs*** and present them separately because they can produce useful work in three cases: (a) when HPC jobs use the time limit as a stopping criterion, (b) when job state is periodically checkpointed to disk, and (c) when a job completes its work before the time limit but fails to terminate cleanly.

Observation 8: *Unsuccessful job terminations in the Google trace are 1.4-6.8x higher than in other traces. Unsuccessful jobs at LANL use 34-80% less CPU time.*

In Figure 3.7, I break down the total number of jobs (left), as well as the total CPU time consumed by all jobs by job outcome (right). First, I observe that the fraction of unsuccessful jobs is significantly higher (1.4-6.8x) for the Google trace than for the other traces. This comparison ignores jobs that timeout for Mustang, because as I explained above, it is unlikely they represent wasted resources. I also note that almost all unsuccessful jobs in the Google trace were aborted. According to the trace documentation [98] these jobs could have been aborted by a user or the

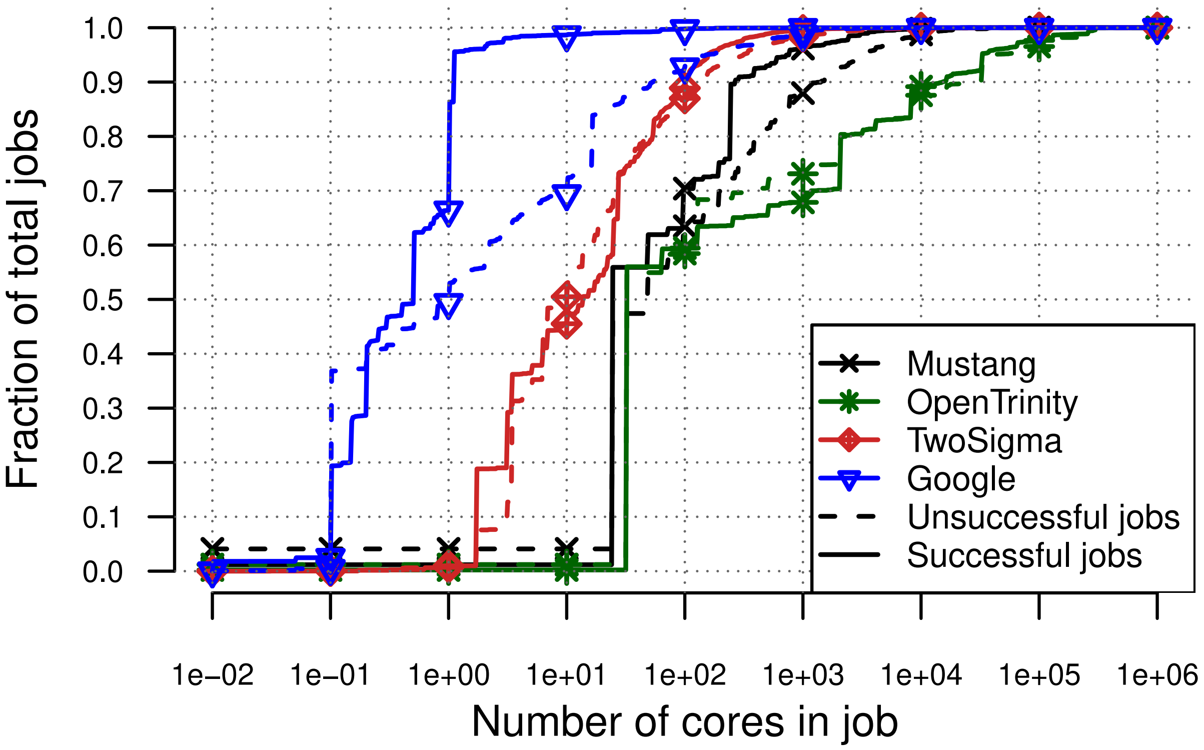


Figure 3.8: CDFs of job sizes (in CPU cores) for unsuccessful and successful jobs.

scheduler, or by dependent jobs that failed. As a result, we cannot rule out the possibility that these jobs were linked to a failure. For this reason, prior work groups all unsuccessful jobs under the "*failed*" label [30], which I choose to avoid for clarity. Another fact that further highlights how blurred the line between failed and aborted jobs can be, is that all unsuccessful jobs in the TwoSigma trace are assigned a failure status. In short, my classification of jobs as "unsuccesful" may seem broad, but it is consistent with the liberal use of the term "failure" in the literature.

I also find that unsuccessful jobs are not equally detrimental to the overall efficiency of all clusters. While the rate of unsuccessful jobs for the TwoSigma trace is similar to the rate of unsuccessful jobs in the OpenTrinity trace, each unsuccessful job lasts longer. Specifically, unsuccessful jobs in the LANL traces waste 34-80% less CPU time than in the Google and TwoSigma traces. It is worth noting that 49-55% of CPU time at LANL is allocated to jobs that time out, which suggests that at least a small fraction of that time may become available through the use of better checkpoint strategies.

Observation 9: *For the Google trace, unsuccessful jobs tend to request more resources than successful ones. This is untrue for all other traces.*

In Figure 3.8, I show the CDFs of job sizes (in CPU cores) of individual jobs. For each trace, I show separate CDFs for unsuccessful and successful jobs. By separating jobs based on their outcome I observe that successful jobs in the Google trace request fewer resources, overall, than unsuccessful jobs. This observation has also been made in earlier work [30, 34], but it does not hold for the other traces. CPU requests for successful jobs in the TwoSigma and LANL traces are similar to requests made by unsuccessful jobs. This trend is opposite to what is seen in older

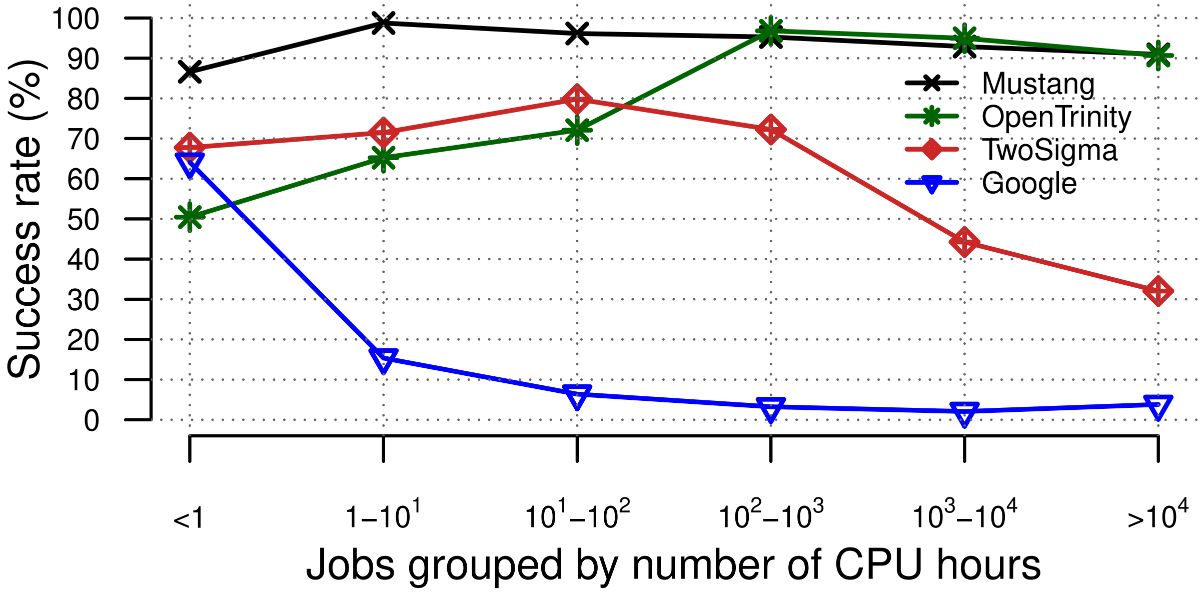


Figure 3.9: Success rates for jobs grouped by CPU hours.

HPC job logs [100], and since these traces were also collected through SLURM and MOAB I do not expect this discrepancy to be due to semantic differences in the way failure is defined across traces.

Observation 10: *For the Google and TwoSigma traces, success rates drop for jobs consuming more CPU hours. The opposite is true for LANL traces.*

For the traces I analyze, the root cause behind unsuccessful outcomes is not reliably recorded. Without this information, it is difficult to interpret and validate the results. For example, I expect that hardware failures are random events whose occurrence roughly approximates some frequency based on the components' Mean Time Between Failure ratings. As a result, jobs that are larger and/or longer, would be more likely to fail. In Figure 3.9 I have grouped jobs based on the CPU hours they consume (a measure of both size and length), and I show the success rate for each group. The trend that stands out is that success rates decrease for jobs consuming more CPU hours in the Google and TwoSigma traces, but they increase and remain high for both LANL clusters. This could be attributed to larger, longer jobs at LANL being more carefully planned and tested, but it could also be due to semantic differences in the way success and failure are defined across traces.

Implications. The majority of papers analyzing the characteristics of job failures in the Google trace build failure prediction models that assume the existence of the trends I have shown on success rates and resource consumption of unsuccessful jobs. Chen et al. [20] highlight the difference in resource consumption between unsuccessful and successful jobs, and El-Sayed et al. [30] note that this is the second most influential predictor (next to early task failures) for their failure prediction models. As I have shown in Figure 3.9, unsuccessful jobs are not linked to resource consumption in other traces. Another predictor highlighted in both studies is job re-submissions, with successful jobs being re-submitted fewer times. I confirm that this trend is consistent across all traces, even though the majority of jobs (83-93%) are submitted exactly once. A final observation that does not hold true for LANL is that CPU time of unsuccessful jobs increases with job runtime [30, 35].

3.6 A case study on plurality and diversity

Evaluating systems against multiple traces enables researchers to identify practical sensitivities of new research and prove its generality. I demonstrate this through a case study on JVuPredict, the job runtime[2] predictor module of the JamaisVu [87] cluster scheduler. The evaluation of JVuPredict with all the traces I have introduced revealed the predictive power of logical job names and consistent user behavior in workload traces. Conversely, I found it difficult to obtain accurate runtime predictions in systems that provide insufficient information to identify job re-runs. This section briefly describes the architecture of JVuPredict (Section 3.6.1) and the evaluation results (Section 3.6.2).

3.6.1 JVuPredict background

Recent schedulers [24, 40, 54, 87, 88] use information on job runtimes to make better scheduling decisions. Accurate knowledge of job runtimes allows a scheduler to pack jobs more aggressively in a cluster [24, 31, 92], or to delay a high-priority batch job to schedule a latency-sensitive job without exceeding the deadline of the batch job. In heterogeneous clusters, knowledge of a job's runtime can also be used to decide whether it is better to immediately start a job on hardware that is sub-optimal for it, let it wait until preferred hardware is available, or simply preempt other jobs to let it run [13, 88]. Such schedulers assume most of the provided runtime information is accurate. The accuracy of the provided runtime is important as these schedulers are only robust to a reasonable degree of error [88].

Traditional approaches for obtaining runtime knowledge are often as trivial as expecting the user to provide an estimate, an approach used in HPC environments such as LANL. As we have seen in Section 3.5, however, users often use these estimates as a stopping criterion (jobs get killed when they exceed them), specify a value that is too high, or simply fix them to a default value. Another option is to detect jobs with a known structure that are easy to profile as a means of ensuring accurate predictions, an approach followed by systems such as Dryad [48], Jockey [33], and ARIA [94]. For periodic jobs, simple history-based predictions can also work well [24, 54]. But these approaches are still inadequate for consolidated clusters without a known structure or history.

JVuPredict, the runtime prediction module of JamaisVu [87], aims to predict a job's runtime when it is submitted, using historical data on past job characteristics and runtimes. It differs from traditional approaches by attempting to detect jobs that repeat, even when successive runs are not declared as repeats. It is more effective, as only part of the history relevant to the newly submitted job is used to generate the estimate. To do this, it uses features of submitted jobs, such as user IDs and job names, to build multiple independent predictors. These predictors are then evaluated based on the accuracy achieved on historic data, and the most accurate one is selected for future

---

[2] The terms *runtime* and *duration* are used interchangeably here.

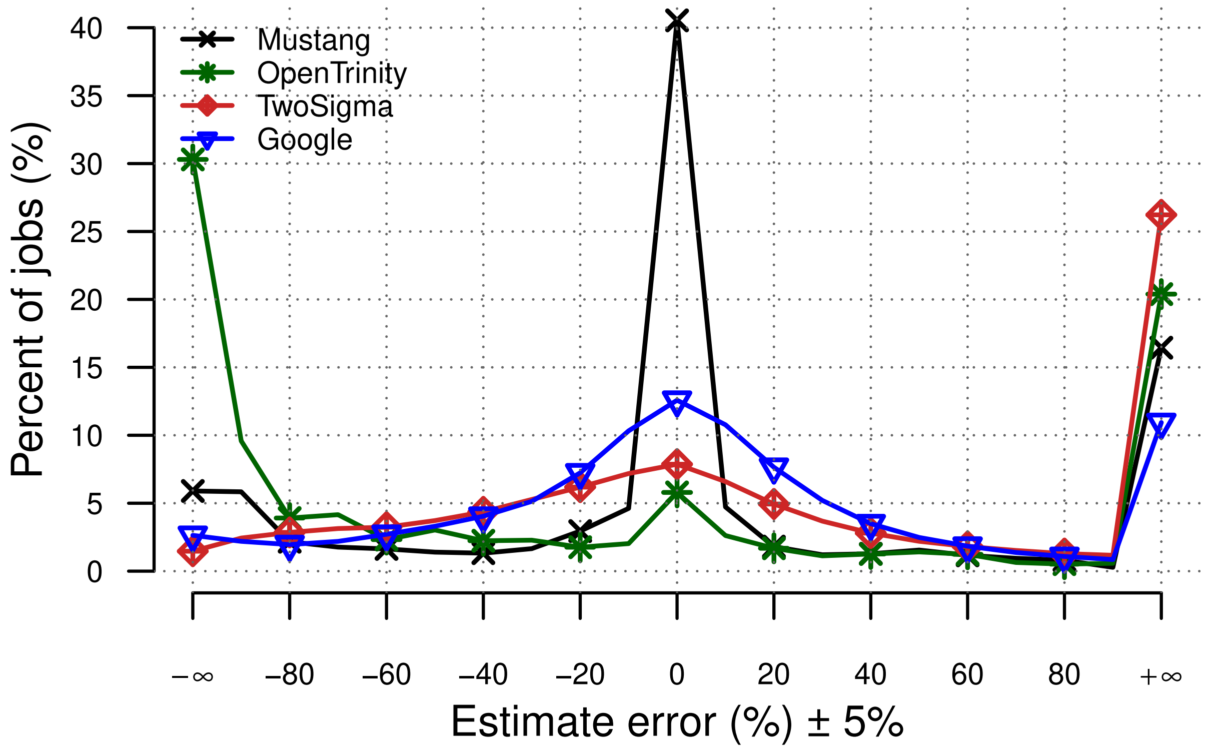


Figure 3.10: Accuracy of JVuPredict predictions of runtime estimates, for all four traces.

predictions. Once a prediction is made, the new job is added to the history and the accuracy scores of each model are recalculated. Based on the updated scores a new predictor is selected and the process is repeated.

3.6.2 Evaluation results

JVuPredict had originally been evaluated using only the Google trace. Although predictions are not expected to be perfect, performance under the Google trace was reasonably good, with 86% of predictions falling within a factor of two of the actual runtime. This level of accuracy is sufficient for the JamaisVu scheduler, which further applies techniques to mitigate the effects of such mispredictions. In the end, the performance of JamaisVu with the Google trace is sufficient to closely match that of a hypothetical scheduler with perfect job runtime information and to outperform runtime-unaware scheduling [87]. This section repeats the evaluation of JVuPredict using the new TwoSigma and LANL traces. The criterion for success is meeting or surpassing the prediction accuracy achieved with the Google trace.

A feature expected to effectively predict job repeats is the job's name. This field is typically anonymized by hashing the program's name and arguments, or simply by hashing the user-defined human-readable job name provided to the scheduler. For the Google trace, predictors using the logical job name field are selected most frequently by JVuPredict due to their high accuracy.

Figure 3.10 shows the evaluation results. On the x-axis I plot the prediction error for JVuPredict's runtime estimates, as a percentage of the actual runtime of the job. Each data point in the plot

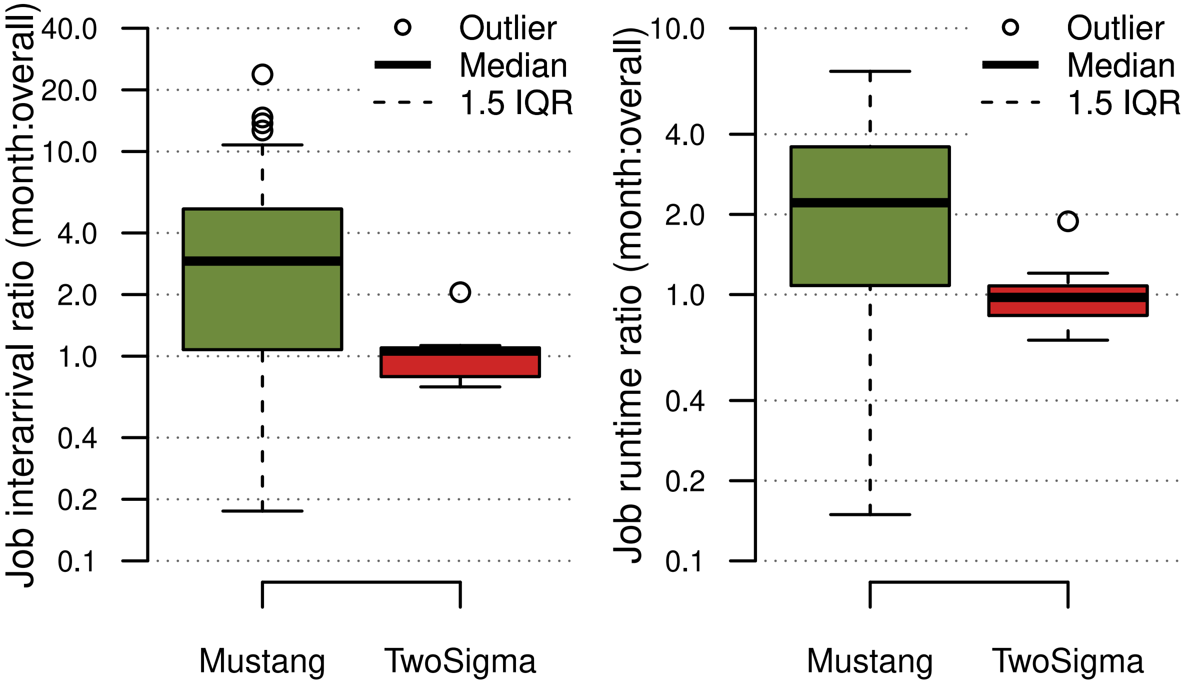


Figure 3.11: *Is a month representative of the overall workload?* The boxplots show distributions of the average job inter-arrival period (left) and duration (right) per month, normalized by the trace's overall average. Boxplot whiskers are defined at 1.5 times the distribution's Inter-Quartile Range (standard Tukey boxplots).

is a bucket representing values within 5% of the nearest decile. The y-axis shows the percentage of jobs whose predictions fall within each bucket. Overestimations of a job's runtime are easier to tolerate than underestimations, because they cause the scheduler to be more conservative when scheduling the job. Thus, the uptick at the right end of the graph is not alarming. For the Google trace, the total percentage of jobs whose runtimes are under-estimated is 32%, with 11.7% of underestimations being lower than half the actual runtime. I mark these numbers as acceptable, since performance of JVuPredict in the Google trace has been proven exceptional in simulation.

Although the logical job name is a feature that performs well for the Google trace, I find it is either unavailable, or unusable in the other traces. This is because of the difficulty inherent in producing an anonymized version of it, while maintaining enough information to distinguish job repeats. Instead, this field is either assigned a unique value for every job, or entirely omitted from the trace. All traces I introduce in this dissertation suffer from this limitation. The absence of the field, however, seems to not affect the performance of JVuPredict significantly. The fields selected by JVuPredict as the most effective predictors of job runtime for the Mustang and TwoSigma traces are: the ID of the user who submitted the job, the number of CPU cores requested by the job, or a combination of the two. I find that the TwoSigma workload achieves identical performance to Google: 31% of job runtimes are underestimated and 15% are predicted to be less than 50% of the actual runtime. The Mustang workload is much more predictable, though, with 38% of predictions falling within 5% of the actual runtime. Still, 16% of job runtimes were underestimated by more than half of the actual runtime. The similarity between the TwoSigma and Mustang results suggests that JamaisVu would also perform well under these workloads. Note that these results extend to the Google trace when the job name is omitted.

OpenTrinity performs worse than every other trace. Even though the preferred predictors are, again, the user ID and the number of CPU cores in the job, 55% of predictions have been underestimations. Even worse, 24% of predictions are underestimated by more than 95% of the actual runtime. A likely cause for this result is the variability present in the trace. I am unsure whether this variability is due to the short duration of the trace, or due to the workload being more inconsistent during the OpenScience configuration period.

In conclusion, two insights were obtained by evaluating JVuPredict with multiple traces. First, I find that although logical job names work well for the Google trace, they are hard to produce in anonymized form for other traces, so they may often be unavailable. Second, I find that in the absence of job names, there are other fields that can substitute for them and provide comparable accuracy for all but the OpenTrinity trace. Specifically, the user ID and CPU core count for every job seem to perform best for both TwoSigma and the Mustang trace.

3.7 On the importance of trace length

Working with traces often forces researchers to make key assumptions as they interpret the data, in order to cope with missing information. A common (unwritten) assumption when using or analyzing a trace, is that it sufficiently represents the workload of the environment wherein it was collected. At the same time the Google trace spans only 29 days, while other traces I study in this dissertation are 3-60 times longer, even covering the entire lifetime of the cluster in the case of Mustang. Being unsure whether 29 days are sufficient to accurately describe a cluster's workload, I decided to examine how representative individual 29-day periods are of the overall workload in the TwoSigma and Mustang traces.

The experiment consisted of dividing the traces in 29-day periods. For each such month I then compared the distributions of individual metrics against the overall distribution for the full trace. The metrics I considered were: job sizes, durations, and interarrival periods. Overall I found consecutive months' distributions to vary wildly for all these metrics. One distinguishable trend, however, is that during the third year the Mustang cluster is dominated by short jobs arriving in bursts.

Figure 3.11 summarizes the results by comparing the averages of different metrics for each month against the overall average across the entire trace. The boxplots show the distributions of average job interarrivals (left) and durations (right) per month, when normalized by the overall average for the trace. The boxplots are standard Tukey boxplots, where the box is framed by the 25^{th} and 75^{th} percentiles, the dark line represents the median, and the whiskers are defined at 1.5 times the distribution's Inter-Quartile Range (IQR), or the furthest data point if no outliers exist (shown in circles here). I see that individual months vary significantly for the Mustang trace, and they differ somewhat less across months in the TwoSigma trace. More specifically, the average job interarrival of a given month can be 0.7-2.0x the value of the overall average in the TwoSigma trace, or 0.2-24x the value of the overall average in the Mustang trace. Average job durations can fluctuate between 0.7-1.9x of the average job duration in the TwoSigma trace, and 0.1-6.9x of the average in the Mustang trace. Overall, the results conclusively show that the cluster workloads display significant differences from month to month.

3.8 Related work

The Parallel Workloads Archive (PWA) [32] hosts the largest collection of public HPC traces. At the time of this writing, 38 HPC traces have been collected between 1993 and 2015. The HPC traces complement this collection. The Mustang trace is unique in a number of ways: it is almost two times longer in duration than the longest publicly available trace, contains four times as many jobs, and covers the entire lifetime of the cluster enabling longitudinal analyses. It is also similar in size to the largest clusters in PWA and its distribution of job duration is shorter than all other HPC traces. The OpenTrinity trace is also complementary to existing traces, as it is collected on a machine almost two times bigger than the largest supercomputer with a publicly available trace (Argonne National Lab's Intrepid) as far as CPU core count is concerned.

Prior studies have looked at private cluster traces, specifically with the aim of characterizing MapReduce workloads. Ren et al. [73] examine three traces from academic Hadoop clusters in an attempt to identify popular application styles and characterize the input/output file sizes, the duration, and the frequency of individual MapReduce stages. These clusters handle significantly less traffic than the Google and TwoSigma clusters I examine. Interestingly, a sizable fraction of interarrival periods for individual jobs are longer than 100 seconds, which resembles my HPC workloads. At the same time, the majority of jobs last less than 8 minutes, which approximates the behavior in the Google trace. Chen et al. [21] look at both private clusters from Cloudera customers and Internet services clusters from Facebook. On the one hand, their private traces cover less than two months, while on the other hand their Facebook traces are much longer than the Google trace. Still, there are similarities in traffic, as measured in job submissions per hour. Specifically, Cloudera customers' private clusters deal with hundreds of job submissions per hour, a traffic pattern similar to the Two Sigma clusters, while Facebook handles upwards of a thousand submissions per hour, which is more related to traffic in the Google cluster. The diversity across these workloads further emphasizes the need for researchers to focus on evaluating new research using a diverse set of traces.

Other studies that look at private clusters focus on Virtual Machine workloads. Shen et al. [80] analyze datasets of monitoring data from individual VMs in two private clusters. They report high variability in resource consumption across VMs, but low overall cluster utilization. Cano et al. [16] examine telemetry data from 2000 clusters of Nutanix customers. The frequency of telemetry collection varies from minutes to days and includes storage, CPU measurements, and maintenance events. The authors report fewer hardware failures in these systems than previously reported in the literature. Cortez et al. [23] characterize the VM workload on Azure, Microsoft's cloud computing platform. They also report low cluster utilization and low variability in tenant job sizes.

3.9 Conclusion

I have introduced and analyzed job scheduler traces from two private and two HPC clusters. The analysis showed that the private clusters resemble the HPC workloads studied, rather than the popular Google trace workload, which is surprising. This observation holds across many aspects of the workload: job sizes and duration, resource allocation, user behavior variability, and unsuccessful job characteristics. I also listed prior work that relies too heavily on the Google trace's characteristics and may be affected.

Finally, I demonstrated the importance of dataset plurality and diversity in the evaluation of new research. For job runtime predictions, I show that using multiple traces allowed us to reliably rank data features by predictive power. I hope that by publishing the traces I will enable researchers to better understand the sensitivity of new research to different workload characteristics.

Chapter 4

3Sigma: a runtime distribution based scheduler

Knowledge of pending jobs' runtimes has been identified as a powerful building block for modern cluster schedulers [24, 54, 88]. With it, a scheduler can pack jobs more aggressively in a cluster's resource assignment plan [24, 54, 88, 92], such as by allowing a latency-sensitive best-effort job to run before a high-priority batch job provided that the priority job will still meet its deadline. Runtime knowledge allows a scheduler to determine whether it is better to start a job immediately on suboptimal machine types with worse expected performance, wait for the jobs currently occupying the preferred machines to finish, or to preempt them [13, 88]. Exploiting job runtime knowledge leads to better, more robust scheduler decisions than relying on hard-coded assumptions.

In most cases, the job runtime estimates are based on previous runtimes observed for similar jobs (e.g., from the same user or by the same periodic job script)—a point estimate (e.g., mean or median) is determined from the relevant history. When such estimates are accurate, schedulers relying on them outperform those using other approaches. Further, previous research [88] has shown that these schedulers can be robust to a reasonable degree of runtime variation (e.g., up to 50%).

However, I find that the estimate errors, while expected in large, multi-use clusters, cover an unexpectedly larger range. Applying a state-of-the-art ML-based predictor [87] to the Google, TwoSigma, and Mustang (Chapter 2) shows good estimates in general (e.g., 77–92% within a factor of two of the actual runtime and most much closer). Unfortunately, 8–23% are not within that range, and some are off by an order of magnitude or more. Thus, a significant percentage of runtime estimates will be well outside the error ranges previously reported.

Worse, I find that schedulers relying on runtime estimates cope poorly with such error profiles. Comparing the middle two bars of Fig. 4.1 shows one example of how much worse a state-of-the-art scheduler does with real estimate error profiles as compared to having perfect estimates.

This chapter describes the 3Sigma cluster scheduling system, which uses all of the relevant runtime history for each job rather than just a point estimate derived from it. Instead, it uses expected runtime *distributions* (e.g., the histogram of observed runtimes), taking advantage of the much richer information (e.g., variance, possible multi-modal behaviors, etc.) to make more robust decisions. The first bar of Fig. 4.1 illustrates 3Sigma's efficacy, showing that it approaches

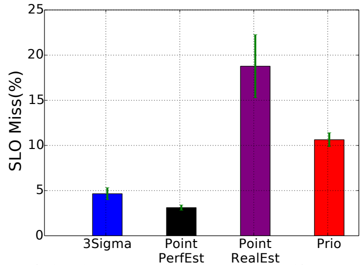


Figure 4.1: Comparison of 3Sigma with three other scheduling approaches w.r.t. SLO (deadline) miss rate, for a mix of SLO and best effort jobs derived from the Google cluster trace [72] on a 256-node cluster. (Details in §4.4.1) 3Sigma, despite estimating runtime distributions online with imperfect knowledge of job classification, approaches the performance of a hypothetical scheduler using perfect runtime estimates (`PointPerfEst`). Full historical runtime distributions and mis-estimation handling helps 3Sigma outperform `PointRealEst`, a state-of-the-art point-estimate-based scheduler (detailed in §4.1.2). The value of exploiting runtime information, when done well, is confirmed by comparison to a conventional priority-based approach (`Prio`).

the hypothetical case of a scheduler with perfect point estimates.

By considering the range of possible runtimes for a job, and their likelihoods, 3Sigma can explicitly consider the various potential outcomes from each possible plan and select a plan based on optimizing the expected outcome. For example, the predicted distribution for one job might have low variance, indicating that the scheduler can be aggressive in packing it in, whereas another job's high variance might suggest that it should be scheduled early (relative to its deadline). 3Sigma similarly exploits the runtime distribution to adaptively address a significant problem with point over-estimates, which may suggest that the scheduler avoid scheduling a job based on the likelihood of missing its deadline.

Full system and simulation experiments with production-derived workloads demonstrate 3Sigma's effectiveness. Using its imperfect but automatically-generated history-based runtime distributions, 3Sigma outperforms both a state-of-the-art point-estimate-based scheduler and a priority-based (runtime-unaware) scheduler, especially for mixes of deadline-oriented jobs and latency-sensitive jobs on heterogeneous resources. 3Sigma *simultaneously* provides higher (1) SLO attainment for deadline-oriented jobs and (2) cluster goodput (utilization). In most cases, 3Sigma performs nearly as well as the hypothetical system with perfect estimates.

This chapter makes four primary contributions. First, it exposes a major problem with applying recent runtime-estimate-guided schedulers to large, multi-use clusters: significant numbers of bad estimates including some large outliers. Second, it describes an approach, which leverages full runtime distributions, that solves this problem as well as an implemented scheduling system (3Sigma) based on this solution. Third, it describes new core scheduler mechanisms, also implemented in 3Sigma, needed to make distribution-based scheduling efficient and scalable—as well as to mitigate the effects of outliers falling outside the observed history. Fourth, it reports on end-to-end experiments on a real 256-node cluster, showing that 3Sigma robustly exploits runtime distributions to improve SLO attainment and best-effort performance, dealing gracefully with the complex runtime variations seen in real cluster environments.

4.1 Background and related work

Cluster consolidation in modern datacenters forces cluster schedulers to handle a diverse mix of workload types, resource capabilities, and user concerns [72, 79, 92]. One result of this has been a resurgence in cluster scheduling research. This section focuses on work related to using information about job runtimes to make better scheduling decisions.

Accurate job runtime information can be exploited to significant benefit in at least three ways at schedule-time.

1) Cluster workloads are increasingly a mixture of business-critical production jobs and best-effort engineering/analysis jobs. The production jobs, often submitted by automated systems [50, 83], tend to be resource-heavy and to have strict completion deadlines [24, 54]. The best-effort jobs, such as exploratory data analytics and software development/debugging, while lower priority, are often latency-sensitive. Given runtime estimates, schedulers can more effectively pack jobs, simultaneously increasing SLO attainment for production jobs and reducing average latency for best-effort jobs [24, 54, 88].

2) Datacenter resources are increasingly heterogeneous, and some jobs behave differently (e.g., complete faster) depending upon which machine(s) they are assigned to. Maximizing cluster effectiveness in the presence of jobs with such considerations can be more effective when job runtimes are known [13, 88, 101].

3) Many parallel computations can only run when all tasks comprising them are initiated and executed simultaneously (gang-scheduling) [65, 68]. Maximizing resource utilization while arranging for such bulk resource assignments is easier when job runtimes are known.

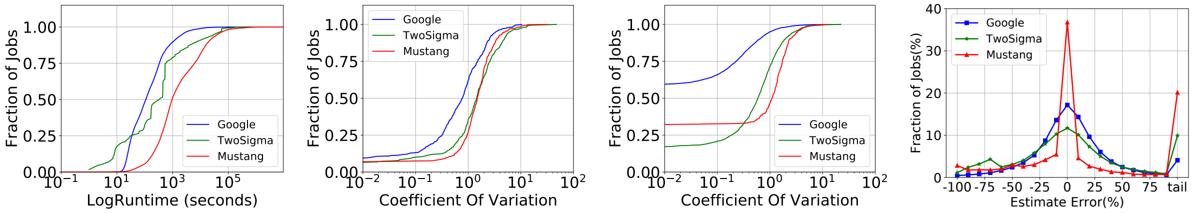
Thus, many recent systems [24, 38, 40, 54, 88] make use of job runtime estimates provided by users or predicted from previous runs of similar jobs. Such systems assume that the predictions are accurate, and they may face severe performance penalties if a significant percentage of runtime estimates is outside a relatively small error range. Worse, I find that this is to be expected in many environments.

4.1.1 Runtime variation and uncertainty

Analysis of job runtime predictability in production environments reveals that consistently accurate predictions should not be expected. Specifically, this section discusses observations from the analysis of job traces from three environments (details in Sec. 2.4). I observe the following:

First, job runtimes are heavy-tailed (longest jobs are much longer than others), suggesting that at least a degree of un-predictability should be expected. Heavy tails can be seen in the distribution of runtimes for each workload (Fig. 4.2(a)).

Second, job runtimes within related subsets of jobs exhibit high variability. I illustrate this with distributions of the Coefficient of Variation (CoV; ratio of standard deviation to mean), within each subset clustered by a meaningful feature, such as user id (Fig. 4.2(b)) or quantity of resources requested (Fig. 4.2(c)). CoV values larger than one (the CoV of an exponential distribution) is typically considered high variability. Large percentages of subsets in each of the workloads have high variability, with more occurring in the TwoSigma and Mustang workloads than in the Google workload.



(a) Runtime CDF (b) CoV - User Id (c) CoV - Resources Requested (d) Estimate Error

Figure 4.2: Analyses of cluster workloads from three different environments: (a) Distribution of job runtimes (b) Distribution of Coefficient of Variation for each subset grouped by user id (c) Distribution of Coefficient of Variation for each subset grouped by amount of resources requested (d) Histogram of Estimate Errors comparing runtime estimates from the state-of-the-art JVuPredict predictor and actual job runtimes. Estimate Error values computed by $\frac{\text{estimate}-\text{actual}}{\text{actual}} \times 100$. Each datapoint is a bucket representing values within 5% of the nearest decile. The "tail" datapoint includes all estimate errors $> 95\%$. Cluster:SC. Workload:Google_E2E, TwoSigma_E2E, MUSTANG_E2E

Third, I evaluate the quality of the estimates from a state-of-the-art predictor and confirm that a significant percentage of estimates are off by factor of two or more. For this evaluation, I generated a runtime estimate for each job and compared with the actual observed runtime in the trace. I use JVuPredict, the runtime predictor module from the recent JamaisVu [87] project to generate runtime estimates. JVuPredict produces an estimate for each job by categorizing jobs (historical and new) using common attributes, such as submitting user or resources requested, and choosing the estimate from the category that has produced the best estimates in the past. Smith et al. [81] describe a similar scheme and its effectiveness for parallel computations.

Fig. 4.2(d) is the histogram of percent estimate error. For all workloads, most job runtimes are estimated reasonably (e.g., ±25% error), but few are perfect. Worse, in each workload, a substantial fraction of jobs are over- or under-estimated by a large margin, well outside the range of errors considered in previous works [40, 88]. Even for the Mustang workload, which has large proportion of jobs with very accurate (±5% error) estimates, at least 23% jobs have estimate error larger than 95% and substantial amount of jobs have estimate error less than -55%. The TwoSigma trace has the fewest jobs with very accurate estimates and many jobs in both tails of the distribution. The Google cluster trace has fewer jobs in the tails of the distribution, but still has 8% of jobs mis-estimated by a factor of two or more.

Overall, I conclude that multi-purpose cluster workloads exhibit enough variability that even very effective predictors will have more and larger mis-estimates than has been assumed in previous research on schedulers that use information about job runtimes.

4.1.2 Mis-estimate mitigation strategies

The scheduling research community has explored techniques to mitigate the effects of job runtime mis-estimates, which can significantly hamper a scheduler's performance.

Some environments (e.g. [54, 95]) use conservative over-provisioning to tolerate mis-estimates by providing the scheduler more flexibility. Naturally, this results in lower cluster utilization, but does reduce problems. Morpheus [54] re-assigns resources to jobs that require more resources at runtime. Not all applications are designed to be *elastic*, though, and some cannot make use of additional resources.

Preemption can be applied to address some issues arising from mis-estimates, like it is used in many systems to re-assign resources to new high-priority jobs, either by killing (e.g., in container-based clusters [95]) or migrating (e.g., in VM-based systems [99]) jobs.

Various other heuristics have been used to mitigate the effects of mis-estimates. [86] addresses mis-estimations of runtimes for HPC workloads by exponentially increasing under-estimated runtimes and then reconsidering scheduling decisions. Other systems [22, 59] use the full runtime distribution to compare the expected benefits of scheduling jobs. The "stochastic scheduler" [77] uses a conservative runtime estimate by padding the observed mean by one or more standard deviations. Such heuristics help (3Sigma borrows the first two), but do not eliminate the problem.

4.1.3 Distribution-based scheduling

Are estimates of job runtime distributions more valuable than point estimates (e.g., estimates of the average job runtime) for cluster scheduling? Intuitively, the distribution provides strictly more information to the scheduler than the point estimate. A simple example below illustrates the point. Suppose two jobs arrive to be scheduled on a toy cluster, and the resources are sufficient to execute only one job at a time. Further, one job is an SLO job with a 15 minute deadline, and the other is a best-effort (BE) job. The objective of the scheduler is to minimize SLO violations, while also minimizing BE job latency. The key question is which job should be executed first?

To answer that question, the scheduler naturally needs more information. Let's start by assuming a point-estimate based scheduler. In our example, imagine that the average runtime of jobs like each of these is known to be 5 minutes. Because the deadline window of 15 minutes is 50% longer the sum of the two point estimates (10 minutes), one might assume that scheduling the BE job first would be relatively safe, which would allow the BE job to start early while still respecting the deadline of the SLO job.

Consider, instead, a distribution-based scheduler, and let's imagine two cases: A and B. In case A, the runtime distribution of each job (SLO and BE) is uniform over the interval 0 to 10 minutes. The average runtime is still 5 minutes, but the scheduler is able to calculate that the probability of the SLO job missing its deadline would be 12.5% if the BE job were scheduled first. Hence, scheduling the SLO job first may be desirable. For case B, in contrast, imagine that the distributions are uniform over the interval 2.5 to 7.5 minutes. Again, the average runtime is 5 minutes, but now the scheduler may safely schedule the BE job first, because even if both jobs execute with worst-case runtimes, the SLO job will finish in the allotted 15 minute window.

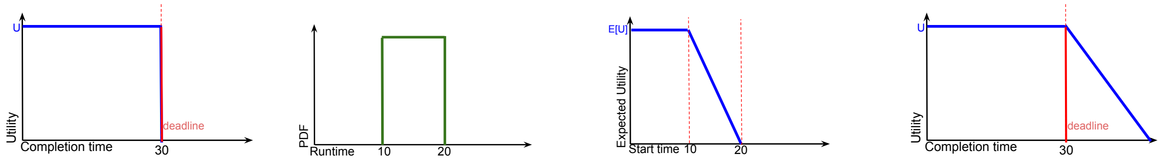
The key observation is that the distributions enable the scheduler to make better-informed decisions; knowing just the average job runtime is not nearly as valuable as knowing whether jobs are drawn from distribution A or B. Two caveats should be mentioned here. First, we implied in this discussion that the SLO deadline is strict; in some environments, this may not be true, and some weighting between BE job start time and SLO miss rate may be desirable. Second, the discussion assumed that the distribution supplied to the scheduler is accurate. In practice, the distribution will have to be estimated in some way—likely from historical job runtime data—and may differ from observed behavior. I address both of these topics later and show that estimated distributions are effective for the workloads studied.

4.2 Distribution-based Scheduling

In this section, I describe the mechanisms that enable schedulers to use the full runtime distribution, as opposed to point estimates. Any scheduler wanting to take advantage of runtime information can use the following generic scheduling algorithm. The scheduler first generates all possible placement options (*resource_type*, *start_time*), each of which has an associated utility. The scheduler chooses to run the set of jobs which both maximize overall sum of utility and fit within the available resources.

Using point runtime estimates, we can find the best schedule using basic optimization techniques, e.g., mixed integer linear programming(MILP). However, with runtime distributions, we have a much larger state-space to consider. For each running job, there are many possible outcomes. Naively considering all scenarios easily makes this problem intractable. Instead of considering each option, I use the *expected utility* per job and *expected resource consumption* over time. This section describes how both of these values are calculated.

4.2.1 Valuation of scheduling options



(a) Job's Utility Function (b) Estimated PDF of Runtime (c) Expected Utility Function (d) Job's Utility Function with Overestimate Handling

Figure 4.3: Example curves for estimating utility for a given job. Each job is associated with a utility function (a) describing its value as a function of completion time. 3σPredict produces a PDF (b) describing potential runtimes for the job. 3σSched combines them to compute expected utility (c) for the job as a function of its start time. [Note the different x-axes for (a), (b), and (c).] As described in §4.3.2, the overestimate handling technique involves modifying the utility function (a) associated with the job with an extended version illustrated in (d).

For each job, there is a set of possible placement options. The placement of the job dictates the job's final completion time, and consequently, it's usefulness or *utility*. A scheduler needs to place jobs in a way that maximizes overall utility. This section describes how to associate job placement options with the utility of the job.

Utility. To make informed placement decisions, a scheduler must quantify its options relative to the success metric the job cares about. I use utility functions to represent a mapping from the domain of possible job placement options and completion times to the potential utility of the job. I assume that a cluster administrator or an expert user will be able to define the utility function on a job-by-job basis. However, in this work, I model the utility of SLO and latency sensitive jobs separately. The utility curve used for SLO jobs is shown in Fig. 4.3(a). This curve models a job with constant utility if completed within the deadline, and zero utility if completed after the deadline. On the other hand, I represent latency sensitive jobs as having a linearly decreasing function over time to declare preference to complete faster.

Expected utility. For each placement option (*resource_type*, *start_time*), a scheduler computes the expected utility of a job using the runtime distribution. The expected utility is calculated as the sum of utilities for each runtime t, weighted by the probability that the job runs for t:

$$E[U(\text{startTime})] = \int_0^{\max(runtime)} U(\text{startTime} + t) PDF(t) dt \tag{4.1}$$

where $U(t)$ is utility function for placement in terms of completion time, and PDF is the probability density function for the job runtime. Fig. 4.3 provides a simple example.

4.2.2 Expected resource consumption

To calculate the set of available resources over time, we need to estimate the resource usage of currently running jobs over time. The use of point estimates for runtimes makes an implicit assumption that resource consumption is deterministic. In contrast, using full distributions acknowledges that resource consumption is, in fact, probabilistic for jobs with uncertain duration. Thus, I calculate the *expected* resource consumption, similarly to expected utility (§4.2.1).

The expected resource consumption of a job at time-slice t is dependent on the probability that the job still uses those resources at (i.e., hasn't completed by) time t. Given PDF(t)—the probability density function of a job's runtime, CDF(t) captures the probability with which the job will complete in at most t time units. The inverse CDF, or $1 - CDF(t)$ then captures the probability with which the job will complete in at least t time units, which is also the probability the job still uses the resources at time t. Thus, expected resource consumption at time t equals to the job's resource demand multiplied by $1 - CDF(t)$.

For running jobs, 3σSched updates the runtime distribution, as it has additional information, namely the fact that the job has been running for some elapsed_time. This enables us to dynamically compute a *conditional* probability density function for the job's expected runtime $P(t|t \geq elapsed\_time)$. This probability update simply renormalizes the original $CDF_{\text{original}}(t)$ and computes the updated probability distribution as follows:

$$1 - CDF_{\text{updated}}(t) = \frac{1 - CDF_{\text{original}}(t)}{1 - CDF_{\text{original}}(\text{elapsed\_time})} \tag{4.2}$$

The amount of available resources in the cluster at time t is then computed by subtracting the aggregate expected resource consumption at time t from the full cluster capacity.

4.3 Design and implementation

This section describes the architecture of 3Sigma (Fig. 4.4). 3Sigma replaces the scheduling component of a cluster manager (e.g. YARN). The cluster manager remains responsible for job and resource life-cycle management.

Job requests are received asynchronously by 3Sigma from the cluster manager (Step 1 of Fig. 4.4). As is typical for such systems, the specification of the request includes a number of attributes, such as (1) the name of the job to be run, (2) the type of job to be run (e.g. MapReduce), (3) the user submitting the job, and (4) a specification of the resources requested.

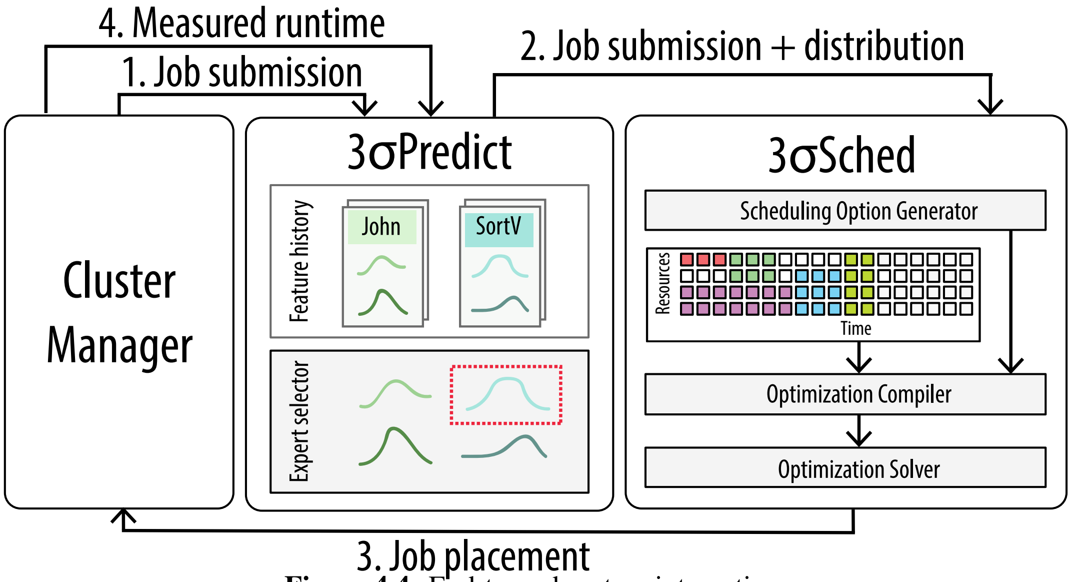


Figure 4.4: End-to-end system integration.

The role of the predictor component, 3σPredict, is to provide the core scheduler with a probability distribution of the execution time of the submitted job. 3σPredict (§4.3.1) does this by maintaining a history of previously executed jobs, identifying a set of jobs that, based on their attributes, are similar to the current job and deriving the runtime distribution the selected jobs' historical runtimes (Step 2 of Fig. 4.4).

Given a distribution of expected job runtimes and request specifications, the core scheduler, 3σSched decides *which jobs to place on which resources and when*. The scheduler evaluates the expected utility of each option (§4.2.1) and the expected resource consumption and availability over the scheduling horizon (§4.2.2). Valuations and computed resource capacity are then compiled into an optimization problem (§4.3.3), which is solved by an external solver. 3σSched translates the solution into an updated schedule and submits the schedule to the cluster manager (Step 3 of Fig. 4.4). On completion, the job's actual runtime is recorded by 3σPredict (along with the attribute information from the job) and incorporated into the job history for future predictions (Step 4 of Fig. 4.4).

In this section, I detail how 3σPredict estimates runtime distributions (§4.3.1), how 3σSched handles mis-estimation (§4.3.2), and the details of the core scheduling algorithm (§4.3.3).

4.3.1 Generating runtime distributions

For each incoming job, 3σPredict provides 3σSched with an estimated runtime distribution. 3σPredict generates this distribution using a black-box approach for prediction. It does not require user-provided runtime estimates, knowledge of job structures, or explicit declarations of similarity to specific previous jobs. However, it does assume that, even in multi-purpose clusters used for a diverse array of activities, most jobs will be similar to some subset of previous jobs.

3σPredict associates each job with set of features. A feature corresponds to an attribute of the job (e.g., user, program name, submission time, priority, resources requested, etc.). Attributes can be combined to form a single feature as well (e.g., user and submission time). 3σPredict tracks job runtime history for each of multiple features, because no single feature is sufficiently predictive for all jobs.

3σPredict associates the new job with historical job runtimes with the same features. Because no single feature is always predictive, 3σPredict generates multiple *candidate distributions* for each job. For example, one candidate distribution may consist of runtimes of jobs submitted by a single user. A second candidate distribution may consist of runtimes of jobs submitted with the same job name.

3σPredict selects one candidate distribution to send to 3σSched. To make this decision, 3σPredict compares each distribution's ability to make accurate point estimates. For a given candidate distribution, 3σPredict makes point estimates in multiple ways as different estimation techniques will be more predictive for different distributions. Specifically, 3σPredict uses four estimation techniques: (a) average, (b) median, (c) rolling (exponentially weighted decay with $\alpha = 0.6$), (d) average of X recent job runtimes. 3σPredict tracks the accuracy of each feature-value:estimator pair, which I refer to as an "expert", using the normalized mean absolute error (NMAE) of past estimates. It designates the runtime distribution from the expert with the lowest NMAE as the distribution estimate of the job.

3σPredict does not make any assumption about the shape of the distribution. Instead, I use empirical distributions, stored as a histogram of the runtimes for each group. Runtimes often exhibit uneven distributions (e.g. heavy-tailed, multi-modal), so I use varying bucket widths to ensure that the shape of the distribution is accurately modeled. I dynamically configure bin sizes using a stream histogram algorithm [11] with a maximum of 80 bins.

Scalability. Storing and querying the entire history of runtimes of a datacenter is not scalable. 3σPredict employs several sketching techniques to greatly reduce the memory footprint. 3σPredict 1) uses a stream histogram algorithm [11] to maintain an approximate histogram of runtimes, 2) computes the average and rolling estimates and NMAE metric for each expert in a streaming manner, and 3) computes the median using recent values as a proxy for the actual median. Using these techniques, 3σPredict provides effective runtime distributions using constant memory, per feature-value.

4.3.2 Handling imperfect distributions

3σPredict estimates the empirical distribution of a job using the history of previously executed jobs. In practice, the estimated runtime distribution is imperfect. Not all jobs have sufficient history to produce a representative distribution. The runtimes of recurring jobs will also evolve over time (e.g. different input data, program updates). 3σSched uses the following mitigation strategies to tolerate error in the estimated runtime distribution.

Under-estimate handling

Distribution schedulers encounter under-estimates when a job runs longer than all historical job runtimes provided in the distribution. An under-estimate can cause a queued job waiting for the busy resource to starve or miss its deadline. To mitigate this, when the elapsed time of the job reaches the maximum observed runtime from the distribution, 3σSched exponentially increments the estimated finish time by 2^t cycles, starting with $t = 0$ in similar fashion to [86]. Exponential incrementing (exp-inc) avoids over-correcting for minor mis-predictions. As 3σSched learns that the under-estimate is more significant, it updates the runtime estimate by progressively longer increments. Note that under-estimates in 3σSched are much more rare compared to using a single point estimate. Point estimate schedulers encounter under-estimates when a job runs longer than the point estimate, whereas 3σSched encounters under-estimates when a job runs longer than all historical job runtimes.

Over-estimate handling

3σSched encounters over-estimates when all historical runtimes are greater than the time to deadline. In this case, the expected utility is zero, leading the scheduler to not see any benefit from spending resources on the job. 3σSched would prefer to keep resources idle, rather than scheduling a job with zero utility. To mitigate the effects of over-estimates, 3σSched proactively changes the utility functions of SLO jobs to degrade gracefully. Instead of a sharp drop to zero utility (Fig. 4.3(a)), 3σSched uses a linearly decaying slope past the deadline (Fig. 4.3(d)). This way, the estimated utility of the job will be non-zero, even if all possible completion times exceed the deadline. The post-deadline utility will be lower than other SLO jobs submitted with the same initial utility. 3σSched will therefore only schedule seemingly impossible jobs when there are available resources in the cluster.

Adaptive over-estimate handling

Enabling 3σSched's over-estimate handling comes at a cost. It increases the number of SLO jobs being tried in favor of completing lower priority jobs. For jobs that were not over-estimated, resources are wasted. Ideally, we should only enable over-estimate handling for jobs which have a reasonable probability of being over-estimates.

3σSched leverages the user provided deadline for SLO jobs in predicting the probability that a job is over-estimated. The deadlines for high priority SLO jobs in production systems are known to be correlated with its actual runtime, since they are usually the result of profiled test runs or previous executions of the same jobs. Thus, 3σSched treats the time from submission to deadline as a reasonable proxy for the upper-bound of the runtime. It compares this upper-bound with the runtime distribution and enables over-estimate handling only if the likelihood of running for less than the upper-bound is below a configured threshold. If the historical runtime distribution implies that the job has no chance of meeting its deadline, even if started immediately upon submission, it is likely that the runtime distribution is skewed toward over-estimation.

4.3.3 Scheduling algorithm

This section describes the details of the core scheduling algorithm used by 3σSched. The discussion includes how I adapt the generalized scheduling algorithm (§4.2) to cope with approximate runtime distributions, the formulation of the optimization problem, and algorithm extensions to support preemption. I conclude the section by examining scalability issues arising from the complexity of solving MILP.

Intuition.

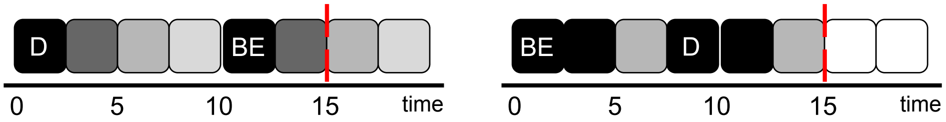
The high level intuition behind the scheduling algorithm is to bin-pack jobs, each represented as a space-time rectangle in cluster resource space-time, where the x-axis represents time and the y-axis enumerates the resources available. Current time is represented as a point on the x-axis. Placing a job further along the x-axis, away from current time, is equivalent to deferring it for future execution. This is useful when a job's preferred resources are busy, but are expected to free up in time to meet the job's deadline. Placing a job on different types of resources (moving its position along the y-axis) changes the shape of the rectangles, as the resource type affects its required resources and completion time. Each job can then be thought of as an enumeration of candidate space-time rectangles, each corresponding to a utility value. The job of the scheduler is to maximize the overall utility of the placement decision by making an instantaneous decision on (a) which jobs to execute and which to defer, and (b) which placement options to pick for those jobs. To achieve this, the scheduler must have a way to formulate all jobs' resource requests so that all pending requests may be considered in aggregate. 3σSched achieves this by formulating jobs' resource requests as Mixed Integer Linear Programming instances.

The scheduler operates on a periodic cycle (at the granularity of seconds, e.g., 1-2s), making a placement decision at each cycle for all pending jobs. The schedule for all pending jobs is re-evaluated every cycle to provide a basic level of robustness to runtime mis-estimation [88]. The sketch of the scheduling algorithm is as follows.
(1) Translate each job's resource request to its MILP representation.
(2) Aggregate jobs' demand constraints.
(3) Construct resource capacity constraints.
(4) Construct the aggregate objective function as the sum of jobs' individual objective functions, modulated by binary indicator variables.
(5) Solve the MILP (using an external MILP solver).
(6) Extract job placement results from MILP solution.
(7) Report the scheduling decision to the resource manager.
(8) Dequeue scheduled jobs from the pending queue.

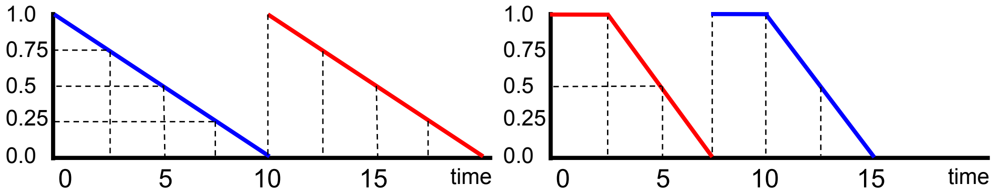
Example

Fig. 4.5 illustrates the example introduced in §4.1.3, where two jobs simultaneously arrive to a single-node cluster: an SLO job (D) with a 15min deadline and a best effort (BE) job. Here, I highlight the mechanics of leveraging the distribution information to achieve the best job schedule. In the left column of Fig. 4.5, I focus on a first scenario, in which both jobs' expected runtimes are drawn from a uniform distribution $U(0,10)$. The right column focuses on a second scenario, where the jobs' expected runtimes are drawn from a uniform distribution $U(2.5,7.5)$. In scenario 1, the scheduler picks a schedule that schedules the SLO job, because it recognizes the risk of it not completing in time otherwise, while in scenario 2, it schedules the BE job first since the SLO job is expected to finish in time regardless of where the runtimes fall within the full distribution. In both cases, it realizes the right decision by maximizing the overall expected utility offered by the two pending jobs.

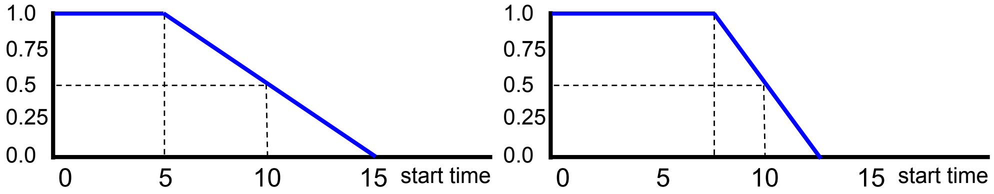
As sketched in §4.3.3, 3σSched first constructs and aggregates the jobs' expected demand.



(a) Scenario 1: Schedule (b) Scenario 2: Schedule



(c) Scenario 1: Inverse CDF(t) (d) Scenario 2: Inverse CDF(t)



(e) Scenario 1: D's Exp. Utility (f) Scenario 2: D's Exp. Utility

Figure 4.5: Job D is an SLO job with a 15min deadline. Job BE is a BE job. Left column: job runtimes $\sim U(0,10)$ (scenario1). Right column: job runtimes $\sim U(2.5,7.5)$ with the same $\mu = 5$ (scenario 2). (a) and (b): The final order that yields maximal utility. The intensity of the black represents the expected resource consumption at the start of each slot (from 100% certainty (darkest) to 25% certainty (lightest), in 25% decrements for the scenario 1 and a 50% decrement for the scenario 2. (c) and (d): Inverse CDF $(1-CDF(t))$, the probability of D (blue) and BE (red) jobs completing before t, which is also the probability of still using the resource at that time. (e) and (f): SLO job's expected utility, set to the probability of the job's completion by the deadline at each start time in this example (note: x-axis is different from other subfigs).

The key insight is that this draw on resources over time is probabilistic (Figs. 4.5(a) and 4.5(b)). E.g., with the SLO job scheduled to start at $t = 0$, it is *expected* to consume the cluster node with 100% probability in the first time slice, and monotonically decreasing probability < 1 in subsequent time slices. Fig. 4.5 plots these probabilities on gray scale from 0 (white: resource not used) to 1 (black: resource expected to be used with 100% probability). I use this grayscale to illustrate the best job schedule in each of the two scenarios in Fig. 4.5. Note that the SLO job meets its 15min deadline in both cases, but is scheduled first in scenario 1 and second in scenario 2. Expected resource consumption is calculated by referring to the inverse CDF (Figs. 4.5(c) and 4.5(d)) of each job's conditional runtime distribution (§4.2.2).

As the scheduler constructs and aggregates resource demands from both jobs, it ensures that their sum does not exceed the expected resource capacity at any given time t in the plan-ahead window $\in [0;20)$. These declarative constraints are automatically generated and added to the MILP problem instance (as described below). The aggregate utility for each job, derived from the expected utility curves, forms the overall objective function to maximize. Figs. 4.5(e) and 4.5(f) show the expected utility curves for the SLO job ("D") in the two scenarios, respectively, and the BE job's utility curves (not shown) are a linearly decaying function with a significantly lower maximum value. Figs. 4.5(a) and 4.5(b) show the best outcome for each of the scenarios. Concretely, I observe that awareness of the runtime distribution allows the scheduler to determine the likelihood of it being safe to delay an SLO job (Fig. 4.5(b)) to minimize the BE job's latency while still meeting the SLO job's deadline.

MILP Formulation

The first step of the algorithm is to convert all jobs to their MILP representation. This is done by generating all possible placement options, both over different types of resources (space) and over time. 3σSched minimizes the set of possibilities by adopting the notion of *equivalence sets*, introduced in [88]. Equivalence sets are sets of resources equivalent from the perspective of a given job, e.g. all nodes with a GPU. 3σSched reasons about these sets of resources instead of enumerating all possible node combinations. As a result, the complexity of MILP depends on the number of equivalence sets rather than the cluster size. Thus, equivalence sets help manage the size of generated MILP in the space dimension. MILP size in the time dimension is controlled by the plan-ahead window *sched*.

A given placement option includes a specification of the equivalence set, the starting time $s \in [now; now + sched]$, the estimated runtime *distribution*, and how many nodes are requested k. The estimated runtime distribution for a running job is reconsidered at every scheduling event, based on how long the job has run so far as described in Eq. 4.2. Updates to the runtime distribution changes the scheduler's expectation of the jobs' future resource consumption. This allows 3σSched to react to mis-estimates, e.g., by re-planning pending jobs waiting for preferred resources to a different set of nodes, or preempting lower priority jobs.

Each placement option is associated with a *const* utility value obtained by using Eq. 4.1. The corresponding objective function for this job becomes a sum of these values modulated by binary indicator decision variables. Namely, given job j and placement option o, the MILP generator associates an indicator variable I_{jo}, adding a constraint that at most one option is selected for each job: $\forall j \sum_o I_{jo} \leq 1$. Thus, the aggregate objective function to maximize is $\sum_j \sum_o U_{jo} I_{jo}$. A solution that maximizes this function effectively selects (a) which jobs to run now, and (b) which placement option o to pick for selected job j.

This objective function is maximized subject to a set of auto-generated constraints: capacity and demand constraints. Demand constraints ensure that (a) the sum of allocations from different resource partitions [88] is equal to the requested quantity of resources k, and (b) at most one placement option is selected: $\forall j \sum_o I_{jo} \leq 1$. Capacity constraints provide the invariant that

$$\forall t \in [now; now + sched] \sum_{jo} k \cdot RC_j(t - s) I_{jo} \leq C(t), \tag{4.3}$$

where $RC_j(t)$ is the expected resource consumption of job j at time t (§4.2.2). This ensures that aggregate allocations do not exceed the expected available capacity $C(t)$ at time t.

MILP Example

Let's examine 3σSched's MILP formulation in detail on the same two job example (Fig. 4.5).

Scenario 1 ($U(0,10)$). First, for each job j, the scheduler generates indicator variables, I_{jt} where $t \in \{0, 2.5, ..., 17.5\}$, that represent whether the job should be scheduled or deferred to each t. The expected utilities of each placement option is also computed. For the SLO job, the expected utility is 1 for all placement options with start times $t \leq 5$. The options that start later will have gradually decreasing values, 0.75, 0.5, 0.25, and 0, as the probability of missing the deadline increases as a function of time. The expected utilities for the BE job will be linearly decreasing values over time.

Second, demand and capacity constraints are generated. A demand constraint $\sum_t I_{jt} \leq 1$ is constructed to ensure at most one option is selected for each job j. Capacity constraints are generated by calculating expected resource consumption for each job. For a job that starts at $t = s$, the expected resource consumption at elapsed_time $= 0, 2.5, 5, 7.5$ is the probability of the job still running and equals 1.0, 0.75, 0.5, 0.25 respectively, zero thereafter.

Third, 3σSched constructs the overall MILP problem by aggregating per-job MILP contributions. Demand constraints are aggregated into the constraints of the problem. Resource capacity constraints are constructed by aggregating the expected resource consumption of placement options across all jobs for each time slot. For example, it is $0.75 I_{SLO;0} + 1.0 I_{SLO;2.5} + 0.75 I_{BE;0} + 1.0 I_{BE;2.5} \leq 1$ for $t = 2.5$.

The aggregate objective function is constructed by adding all $U_{jt} I_{jt}$ terms, where U_{jt} is the expected utility of the placement option for job j that starts at t. Examining the objective function while satisfying the demand and capacity constraints, 3σSched decides to schedule the SLO job first. Starting the SLO job at $t = 10$ would only yield an expected utility of 0.5, which corresponds to a 50% probability of meeting the deadline. The BE job utility gain would be insufficient to offset the SLO utility loss.

Scenario 2 ($U(2.5,7.5)$). First, all the decision variables are generated similarly to the scenario 1. The expected utility is calculated for both jobs. For the SLO job, the expected utility is 1 for all placement options with start times $t \leq 7.5$. The expected utility of the BE job and the demand constraints are same as before.

For a job that starts at $t = s$, the expected resource consumption at elapsed_time $= 0, 2.5, 5$ is $1.0, 1.0, 0.5$ respectively, and zero otherwise.

The aggregate MILP problem is constructed, aggregating demand constraints, objective functions, and generating capacity constraints that ensure the sum of aggregate resource demand does not exceed total resource capacity. This ensures the second job is deferred to start at $t = 7.5$.

Examining the objective function while satisfying the demand and capacity constraints, 3σSched decides to schedule the BE job first and postpones the SLO job, as it is possible to complete both jobs before the deadline. The aggregate expected utility reflects that, as it yields the SLO job utility of 1 when started by $t = 7.5$ and the highest value for the BE job when started at $t = 0$.

Preemption

In rare cases, 3σSched needs to re-consider scheduling decisions for currently running jobs. For example, due to under-estimates, the scheduler may incorrectly choose to aggressively postpone SLO jobs to complete more BE jobs. The scheduler might be able to reschedule if there are enough resources. However, sometimes the only way to meet the deadline is to preempt lower-priority jobs running in the cluster.

Preemption is naturally supported in the existing MILP generation framework, as it is able to simultaneously consider pending jobs for placement and running jobs for preemption. Thus, the goal of the scheduler is to maximize the aggregate value of placed jobs, while incurring a cost for preempting jobs. The latter is a $\sum_r P_r I_r^p$, where I_r^p is an indicator variable tracking whether to preempt a running job r. P_r is the preemption cost for the running job r and is configured by the preemption policy. The overall objective function then becomes $\sum_{j,o} U_{jo} I_{jo} - \sum_r P_r I_r^p$.

The capacity constraint extension, intuitively, credits back resources associated with preempted jobs: $\sum_{jo} k_o RC_j(t-s) I_{jo} \leq C(t) + \sum_r k_r RC_r(t-e) I_r^p$. RC_r is the up-to-date expected resource consumption of the running job r, and e is the elapsed time of r.

Scalability

Solving MILP is known to be an NP-hard problem. To minimize the excessive latency caused by the solver, I apply a number of optimizations. The primary optimization I perform is seeding each new cycle's MILP problem with the solution from the previous cycle. Intuitively, the previous cycle's solution corresponds to leaving the cluster state unchanged. As such, it represents a feasible solution. Second, I have empirically found that the solver spends most of the time validating optimality for the solution it otherwise quickly finds. Thus, I get near-optimal performance by querying the solver for the best solution found within a configurable fraction of its scheduling interval. Third, the plan-ahead window bounds the complexity of the MILP problem by adjusting the range of time over which job placements are considered. Fourth, 3σSched performs some internal pruning of generated MILP expressions, which include eliminating terms with zero constant.

4.4 Evaluation

This section evaluates 3Sigma, yielding five key takeaways. First, 3Sigma achieves significant improvement over the state-of-the-art in SLO miss rate, best-effort job goodput, and best-effort latency in a fully-integrated real cluster deployment, approaching the performance of the unrealistic `PointPerfEst` in SLO miss rate and BE latency. Second, all of the 3σSched component features are important, as seen via a piecewise benefit attribution. Third, estimated distributions are beneficial in scheduling even if they are somewhat inaccurate, and such inaccuracies are better handled by distribution-based scheduling than point-estimate-based scheduling. Fourth, 3Sigma performs well (i.e., comparably to `PointPerfEst`) under a variety of conditions, such as varying cluster load, relative SLO job deadlines, and prediction inaccuracy. Fifth, I show that the 3Sigma components (3σPredict and 3σSched) can scale to $>$10000 nodes.

4.4.1 Experimental setup

I conduct a series of end-to-end experiments and microbenchmarks to evaluate 3Sigma, integrated with Hadoop YARN [89]–a popular open source cluster scheduling framework. I find YARN's support for time-aware reservations and placement decisions and its popularity in enterprise a good

System	Runtime Estimation	Overestimate Handling
3Sigma	Real Distributions	ADAPTIVE
`PointPerfEst`	Perfect Point Estimates	NO
`PointRealEst`	Real Point Estimates	NO
`Prio`	N/A	N/A

Table 4.1: Scheduler approaches compared.

fit for my needs. I implement a proxy scheduler wrapper that plugs into YARN's ResourceManager and forwards job resource requests asynchronously to 3Sigma. Jobs are modeled as Mapper-only jobs. I use a synthetic generator based on Gridmix 3 to generate Mapper-only jobs that respect the runtime parameters for arrival time, job count, size, deadline, and task runtime from the pre-generated trace.

Cluster configurations. I conduct experiments on two cluster configurations: a 256-node real cluster (RC256) and a simulated 256-node cluster (SC256). RC256 consists of 257 physical nodes (1 master + 256 slaves in 8 equal racks), each equipped with 16GB of RAM and a quad-core processor. The simulations complete in $\frac{1}{5}^{th}$ the time on a single node, allowing us to evaluate more configurations and longer workloads. I also conduct an experiment with a simulated 12,583-node cluster (GOOGLE) to evaluate 3Sigma's scalability.

Systems compared. I compare the four scheduler approaches in Table 4.1. 3Sigma is my system in which 3σSched is given real runtime distributions provided by 3σPredict and uses adaptive overestimate handling. Both `PointPerfEst` and `PointRealEst` use an enhanced version of [88] with under-estimate handling (§4.3.2) and preemption (§4.3.3). It represents the state-of-the-art in schedulers that rely on point estimates. This includes Rayon, Morpheus, and TetriSched [24, 54, 88], enhanced with the state-of-the-art in techniques for handling imperfect estimates. `PointPerfEst` is a hypothetical system in which the scheduler is given a correct runtime for every incoming job. `PointRealEst` uses point runtime estimates from 3σPredict. `Prio` is a priority scheduler, giving SLO jobs strict priority over BE jobs rather than leveraging runtime information, which represent schedulers like Borg [95].

Workloads. The bulk of the experiments use workloads derived from the Google trace [72]. I use a Google trace-derived workload (termed ”E2E”) for overall comparisons among schedulers as well as workloads that vary individual workload characteristics (e.g., runtime variation or cluster load) to explore sensitivities. All workloads are 5 hours in length (˜1500 jobs) except for the 2hr E2E (˜600 jobs), used to expedite the experiment in RC256. The E2E workload is synthetically generated from Google trace characteristics. I evaluated the quality of estimates (as in §4.1.1) and confirmed that the runtime predictability of the generated workload was similar to the original Google trace. In simulation, I have also obtained similar experimental results by drawing random trace samples from the original instead of using the E2E workload. To generate a workload, all jobs larger than 256 nodes were filtered out. The remaining jobs — clustered using k-means clustering on their runtimes. I derive parameters for the distributions of the job attributes (e.g., runtime and number of tasks) and the probability mass function of features in each job class. The arrival process used was exponential with a coefficient of variance of 4 (c_a^2=4). I draw jobs from each job class proportionally to the empirical job-class distribution. I also pick job attributes and features for each job according to the empirical distribution of attributes and features from the job-class. Each workload consists of an even mixture of SLO jobs with deadlines and latency sensitive best effort (BE) jobs. SLO jobs have soft placement constraints (preferred resources set to a random 75% of the cluster, as observed in the original trace). SLO jobs run 1.5x longer if scheduled on non-preferred resources.

For the experiment in §4.4.2, I also use workload TwoSigma_E2E and MUSTANG_E2E derived from the TwoSigma and Mustang traces, respectively. For these workloads I filtered out jobs larger than 256 nodes, but took a 5 hour segment of the original workload instead of deriving parameters and regenerating based on the distribution. The segment was randomly selected among many segments that have a similar load to the E2E workload.

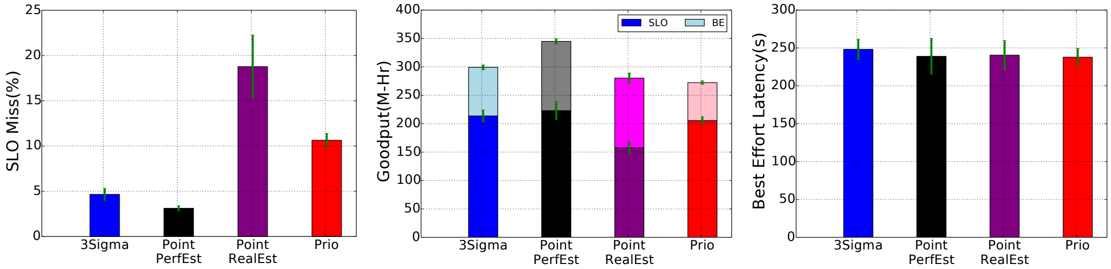
Estimates. Because the experiments are 5-hour windows, I pre-train 3σPredict before running them to produce steady-state estimates for 3Sigma and `PointRealEst`. For the Google workload, I use a subset of the generated trace for pre-training and use the rest for the experiments. Only the features present in the original trace were used to generate point and distribution estimates (e.g., job class, the runtime class membership feature not present in the original trace, was never used in order to maintain a fair experimental setup). For other workloads, I pre-train on jobs completed before the selected 5 hour segment begins.

Workload configurations. For SLO jobs, the deadline slack is an important consideration. Since the original workloads do not include deadline information, I generate deadlines for each SLO job as follows. Deadline slack is defined as $(deadline - submissiontime - runtime)/runtime * 100$ (i.e., a slack of 60% indicates that the scheduler has a window 60% longer than the runtime in which to complete the job). Tighter deadlines are more challenging for schedulers. By default, I select each job's deadline slack randomly from a set of 4 options: 20%, 40%, 60%, and 80%. These default values are much smaller than experimented in [88] (which used slacks of 250% and 300%), matching the finding in [54] that tighter deadlines are also possible.

Load is a measure of offered work ($machine \times hours$) submitted to the cluster scheduler as a proportion of cluster capacity. The nominal offered load is 1.4 (unless specified otherwise). I first chose the load for SLO jobs as 0.7, approximating the load offered by production jobs in [72]. I added equal proportion of BE jobs as to not unfairly bias the scheduling problem towards SLO jobs and to demonstrate the behavior of system under stressful conditions.

Note my definition of load is different from effective load, a ratio of actual resources allocated for all jobs (successful and not successful) to the cluster capacity. Effective load is different for each scheduling approach as they make different allocation decisions, even if the same jobs are injected to the system. In all experiments and for all scheduling approaches, the cluster was run close to its space-time capacity.

Success metrics. I use the following goodness metrics when comparing schedulers. The primary goal is to minimize SLO miss rate: the percentage of SLO jobs that miss their deadline. I also want to measure the total work completed in machine-hours (goodput), showing how much aggregate work is completed, since BE goodput and the goodput of incomplete SLO jobs is not represented by the SLO miss rate. Finally, I measure mean BE latency—the mean response time for BE jobs.



(a) SLO Miss rate (b) Goodput (c) BE latency

Figure 4.6: Compares the performance of 3Sigma with other systems in the real cluster. 3Sigma constantly outperforms `PointRealEst` and `Prio` on SLO miss-rate and Goodput while nearly matching `PointPerfEst`. Cluster:RC256. Workload:E2E

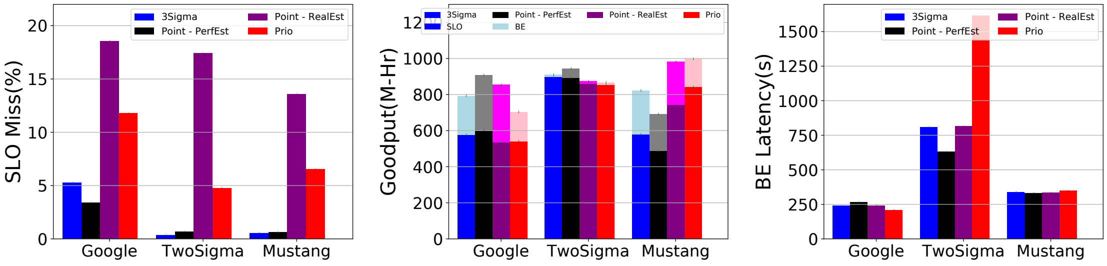
4.4.2 End-to-end performance

Fig. 4.6 shows performance results for the four scheduling systems running on the real cluster (RC256).

3Sigma is particularly adept at minimizing SLO misses, my primary objective, and completing more useful work, approaching `PointPerfEst` and significantly outperforming the non-hypothetical systems. 3Sigma performs well, despite not having the luxury of perfect job runtime knowledge afforded to `PointPerfEst`. It uses historical runtime distributions to make informed decisions, such as whether to start a job early to give ample time for it to complete before its deadline, or to be optimistic and schedule the job closer to the deadline. However, 3Sigma is not perfect. It misses a few more SLO job deadlines than `PointPerfEst`, and it completes fewer best-effort jobs because 3σSched preempts more best-effort jobs to make additional room for SLO jobs for which the distribution indicates a wider range of possible runtimes for a job . BE latency is similar across all system.

`PointRealEst` exhibits much higher SLO miss rates (18%, or 4.0X higher than 3Sigma), and lower goodput (5.4% lower than 3Sigma), because previous approaches struggle with realistic prediction error profiles. Because `PointRealEst` schedules based on only point estimates (instead of complete runtime distributions) and lacks an explicit overestimate handling policy, it makes less informed decisions and struggles to handle difficult-to-estimate runtimes (e.g., due to greater variance for a job type). For underestimated SLO jobs (that ran shorter in the past on average), `PointRealEst` is often too optimistic and starts the job later than it should. For overestimated SLO jobs, `PointRealEst` is often too conservative, neglecting to schedule SLO jobs which are predicted to not finish in time, even if cluster resources are available.

`Prio` misses 12% of SLO job deadlines (2.3x more than 3Sigma). It does not take advantage of any runtime information, thereby missing opportunities to wait for preferred resources or exploit one job's large deadline slack to start a tighter deadline job sooner. `Prio` is better than `PointRealEst` in terms of SLO misses but much worse in BE goodput, as it always prioritizes SLO jobs at the expense of increased preemption of BE jobs, even when deadline slack makes preemption unnecessary. When the runtime is over-estimated, `PointRealEst` may not even attempt to run a job thinking that it would not complete in time, while `Prio` will always attempt to schedule any SLO jobs if there are enough resources.



(a) SLO Miss rate (b) Goodput (c) BE latency

Figure 4.7: Compares the performance of 3Sigma with other systems under workloads from different environments in simulated cluster. 3Sigma constantly outperforms `PointRealEst` and `Prio` on SLO miss rate and Goodput while nearly matching `PointPerfEst`. The Google workload is 5hr variant of E2E. Cluster:SC256. Workload:E2E, TwoSigma_E2E, MUSTANG_E2E

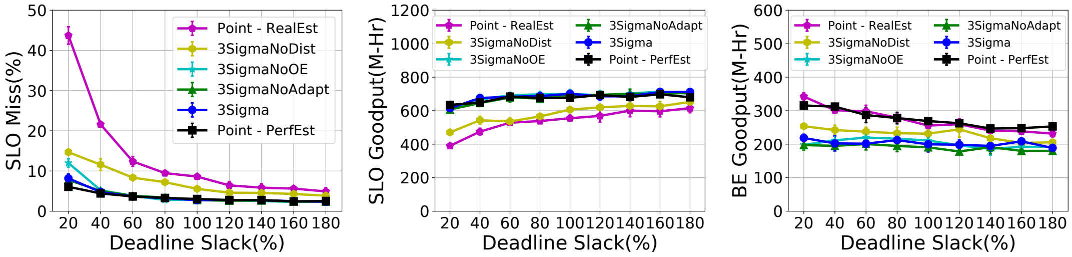
Simulator experiments. I validate the simulation setup (SC256) by running the identical workload to that in experiment in Fig. 4.6. Similar trends are observed across all my systems and success metrics. Table 4.2 shows the small differences observed for the 12 bars shown in Fig. 4.6.

Performance comparison varying workload. Fig. 4.7 summarizes the performance of the scheduling systems under three different workloads. I observe that the overall behavior of the schedulers is similar to my observations in §4.4.2. For all workloads, 3Sigma outperforms `PointRealEst` and `Prio`, while approximately matching the performance of `PointPerfEst`. Surprisingly, for the TwoSigma and Mustang workloads, 3Sigma slightly outperforms `PointPerfEst`. This is possible because, while `PointPerfEst` does receive perfect runtime knowledge as jobs arrive, it does not possess knowledge of future job arrivals (nor do any of the other systems). Consequently, it may make sub-optimal scheduling decisions, such as starting a SLO job late and not leaving sufficient resources for future arrivals. 3Sigma also does not possess knowledge of future job arrivals, but it tends to start SLO jobs earlier than `PointPerfEst` when the distribution suggests likelihood of a runtime longer than the actual runtime.

I also observe that `PointRealEst` performs poorly on SLO miss rate across different workloads. Further, miss-rate is only slightly better for Mustang. This is surprising, as a much larger portion (compared to other workloads) of jobs in Mustang have very accurate point estimates (Fig. 4.2(a)). I believe `PointRealEst` still performs poorly as a small number of the estimates are off by a large margin, adversely affecting the ability of the scheduler to make informed decision. But, many of the mis-estimates are associated with small jobs; consequently, `PointRealEst` and `Prio` are able to provide high goodput despite having high SLO miss-rates.

Metric(unit)	Δ SLO miss(%)	Δ goodput(M-Hr)	Δ BE latency(s)
3Sigma	0.2875	27.10	11.08
`PointPerfEst`	0.6784	25.27	7.282
`PointRealEst`	2.025	22.83	2.383
`Prio`	1.853	19.83	12.07

Table 4.2: Absolute performance difference between real and simulation experiments. Workload:E2E.



(a) SLO Miss (b) SLO Goodput (c) BE Goodput

Figure 4.8: Attribution of Benefit. The lines representing 3Sigma with individual techniques disabled—demonstrating that all are needed to achieve the best performance. The workload is E2E with a constant deadline slack. Cluster:SC256 Workload:DEADLINE-n where $n \in [20,40,60,80,100,120,140,160,180]$

4.4.3 Attribution of benefit

3σSched introduces distribution-based scheduling and adaptive overestimate handling to robustly address the effects of runtime uncertainty. This section evaluates the individual contributions of these techniques. Fig. 4.8 shows performance as a function of deadline slack for 3Sigma, `PointPerfEst`, `PointRealEst`, and three versions of 3Sigma, each with a single technique disabled: `3SigmaNoDist` uses point estimates instead of distributions, `3SigmaNoOE` turns off the overestimate handling policy, and `3SigmaNoAdapt` turns off just the adaptive aspect of the policy and uses maximum overestimate handling for every job.

When the scheduler explicitly handles overestimates (compare `3SigmaNoDist` to `PointRealEst`), SLO miss rate decreases because over-estimated SLO jobs are optimistically allowed to run, rather than discarding them as soon as they appear to not have enough time to finish before the deadline. However, SLO miss rate for `3SigmaNoDist` is still high, because the lack of distribution awareness obscures which jobs are more likely to succeed if tried; therefore, `3SigmaNoDist` wastes resources on SLO jobs that won't finish in time.

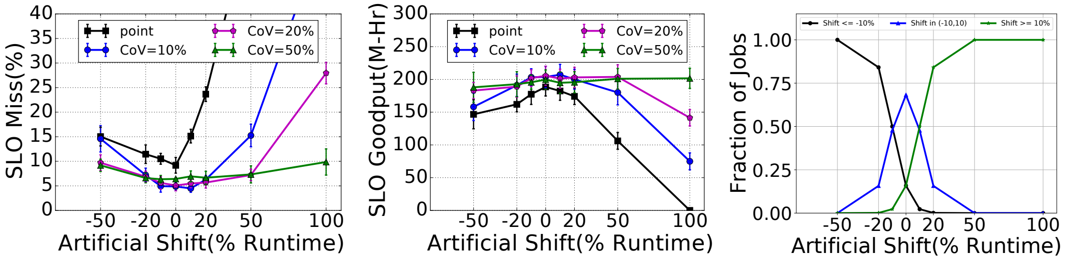
Simply using distribution-based scheduling (see, e.g., `3SigmaNoOE`) drops SLO miss rate to the level of `PointPerfEst` for most deadline slacks. By considering the variance of job runtimes, the scheduler can conservatively schedule jobs with uncertain runtimes and optimistically attempt jobs that are estimated to have a non-zero probability of completion.

Blindly turning on overestimate handling decreases SLO miss rates at the lowest deadline slacks (`3SigmaNoAdapt`). However, `3SigmaNoAdapt` is overly optimistic— even attempting jobs that would seem impossible given their historical runtimes— provided there are enough resources for SLO jobs in the cluster. This over-optimism results in lower BE goodput relative to 3Sigma's adaptive approach of enabling overestimate handling only for a small proportion of the jobs whose distributions indicate likely success.

4.4.4 Distribution-based scheduling benefits

This section explores the robustness of 3Sigma to perturbations of the runtime distribution. In this study, for each job drawn from the E2E workload, I provide 3σSched with a synthetically generated distribution instead of the distribution produced by 3σPredict.

I adjust the synthetic distributions in two dimensions, corresponding to an off-center mean and different variances. The former is realized by artificially shifting the entire distribution by an



(a) SLO Miss rate (b) SLO Goodput (c) Shift Profile

Figure 4.9: 3Sigma's performance when artificially varying runtime distribution shift (x-axis) and width (Coefficient of Variation curves). The runtime distribution provided to the scheduler is $\sim \mathcal{N}(\mu = job\_runtime * (1 + \frac{x}{100}), \sigma = job\_runtime * CoV)$. Each trace consists of jobs that are either within 10% accuracy or under- or over-estimates jobs. The group of jobs achieves a target average artifical shift. (c) shows the breakdown of these job types for each artificial shift value. Distribution-based schedulers always outperforms the point estimate-based scheduler. Tighter distributions perform better than wider distribution with a smaller artificial shift, but wider distributions are better with a larger artificial shift. The workload is 2 hrs in length. Cluster:SC256. Workload:E2E

amount equal to a selected percent difference between the mean of the distribution and the actual runtime. The latter is represented by the CoV, which refers to the ratio of standard deviation to the actual runtime of the job. For each job, the artificial distribution is $\sim \mathcal{N}(\mu = job\_runtime * (1 + shift), \sigma = job\_runtime * CoV)$, where the shift itself is $\sim \mathcal{N}(\mu = shift, \sigma = 0.1)$.

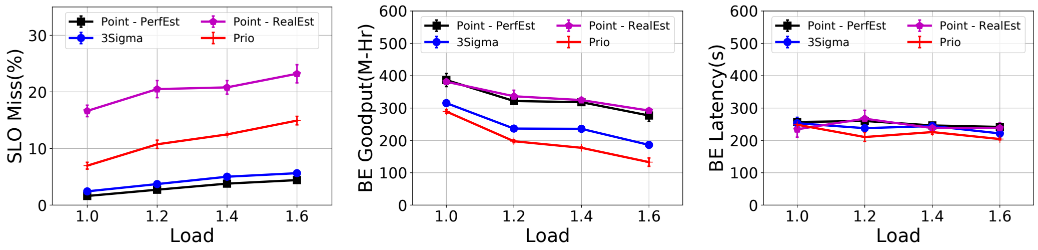
Fig. 4.9 shows the results. Comparing point estimates (`point`) and distribution estimates, I observe that it is strictly better to use distribution estimates (`CoV=x%`) than to use point estimates (`point`) for scheduling jobs. Even at an artificial shift$= 0.0$, where $\approx 70\%$ of estimates are generally accurate (within $\pm 10\%$ error), using a distribution yields 2X fewer SLO misses compared to the point estimates. Hence, even a small proportion of jobs with inaccurate estimates can cause the scheduler to make mistakes and miss the opportunity to finish more jobs on time. Comprehending entire distributions enables the scheduler to reason about uncertainty in runtimes.

Furthermore, for small artificial shifts (within $\pm 20\%$), it is better to have narrower distributions with a smaller CoV. This is because a wider distribution indicates greater likelihood of runtimes that are much shorter and much larger than the actual runtime. The scheduler is more likely to incorrectly make risky decision to start some jobs later than it should and make overly conservative decisions for other jobs.

However, if the actual runtime is far away from the center of the runtime distribution (larger artificial shift), wider distributions provide a benefit. As the distribution widens, the scheduler correctly assigns higher expected utility to scenarios that hedge the risk of runtimes being farther away from the mean. On the other hand, narrower distributions suffer more as the artificial shift deviates further from zero. The likelihood of the job running for the actual runtime decreases significantly, and causes the scheduler to discount the placement options that hedge the associated risks.

4.4.5 Sensitivity analyses

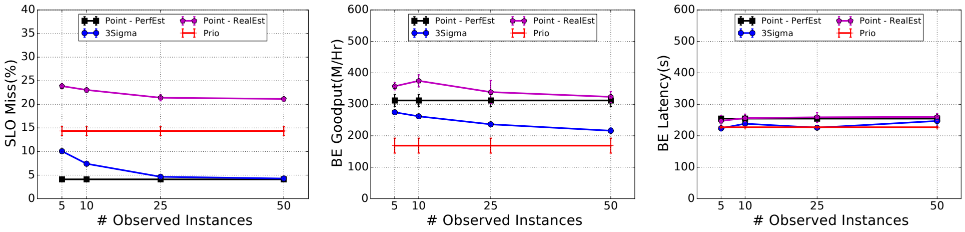
Sensitivity to deadline slack. Fig. 4.8 shows performance as a function of deadline slack. I make two additional observations. First, smaller slack makes it harder to meet SLOs across all policies, due to increased contention for cluster space-time, leading to higher SLO miss rates. Second, best effort goodput decreases for all systems, but for different reasons. As slack increases, `PointPerfEst` sees more wiggle room for placement and tries (and completes) more difficult larger SLO jobs. Since the schedule is optimally packed, it needs to bump best effort jobs in order to schedule more SLO jobs. BE goodput of `PointRealEst` shows similar trends; `PointRealEst` tries more over-estimated jobs, since increasing slack reduces the number of seemingly impossible jobs. 3Sigma, on the other hand, was already trying most completable overestimated jobs, so it sees the smallest decrease in BE goodput. More of the SLO jobs succeed though.



(a) SLO Miss-rate (b) BE Goodput (c) Best Effort latency

Figure 4.10: 3Sigma outperforms others on SLO misses for a range of loads, matching `PointPerfEst` closely. All systems prioritize SLO jobs by sacrificing BE jobs when load spikes. Cluster:SC256, Workload: E2E-LOAD-ℓ where $\ell \in [1.0, 1.2, 1.4, 1.6]$

Sensitivity to load. Fig. 4.10 shows performance as a function of load. As load increases, I observe an increase in all systems' SLO miss rates due to increased contention for cluster resources. The relative effectiveness of `PointPerfEst`and the three realistic scheduling approaches is consistent across the range. I observe that as the load increases, all systems increasingly prioritize SLO jobs, decreasing BE goodput. The gap between the BE goodputs of `PointPerfEst` and 3Sigma widens as 3Sigma makes more room for each incoming SLO job to address its uncertainty about runtimes.



(a) SLO Miss-rate (b) BE Goodput (c) BE latency

Figure 4.11: 3Sigma outperforms others on SLO Misses for a range of runtime variability. 3Sigma matches `PointPerfEst` in terms of SLO misses at the sacrifice of Best Effort goodput. Cluster:SC256. Workload:E2E-SAMPLE-n where $n \in [5, 10, 25, 50, 75, 100]$

Sensitivity to sample size. Another concern may be: how is the performance of the scheduler affected by the number of samples observed per feature (user, job names, etc.)? To answer this question, I used another modified version of the E2E workload where I controlled the number of samples comprising the distributions used by 3Sigma, drawing those samples from the original distributions. I also created a version of `PointRealEst` where the point estimates were derived from the observed samples. In Fig. 4.11, I vary the number of samples used from 5 to 100. I observe that increasing the number of samples from 5 to 25 significantly improved performance (for both schedulers), but by 25 samples, the performance of 3Sigma converges to the performance of `PointPerfEst`. 3Sigma outperforms `PointRealEst`at each point and benefits more from additional instances, since it uses the distribution rather than just the mean. Naturally, `PointPerfEst` and `Prio` are not affected.

4.4.6 Scalability

This section shows that 3Sigma can handle the additional complexity from distribution-based scheduling even while managing more than 12500 nodes and a job submission rate comparable to the heaviest load observed in the Google cluster trace (3668 jobs per hour).

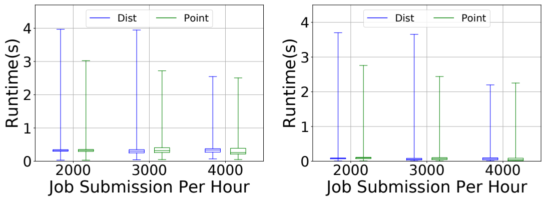
3Sigma requires more CPU time to make decisions than not using runtime estimates (e.g., `Prio`), which can affect scheduler scalability. Although previous work [24, 88] has shown that packing cluster space-time using runtime estimates can be sufficiently efficient for 100s to 1000s of nodes, 3Sigma adds sources of overhead not evaluated in such previous work: (1) latency of 3σPredict at Job Submission (I/O and computation for looking up the correct group of jobs in the runtime history database and generating distribution), (2) latency from additional computation (e.g. computing expected utility and expected resource consumption) to formulate the bin-packing problem, and (3) increased solver runtime due to increased complexity of the bin-packing problem at 3σSched.

In this experiment, 3Sigma schedules microbenchmark workloads, SCALABILITY-*n*. Each workload consists of *n* jobs per hour for 5 hours. The ratio of tasks to job matches those observed in the Google cluster trace. The load is set to 0.95. Even under these conditions, the latency of producing distributions at 3σPredict is negligible (maximum=14ms) compared to the job runtimes in the trace. 3σPredict maintains minimal state for each group of jobs, so the cost of data retrieval is low. Similar latency is observed for producing point estimates, since most of the work is the same (accessing histories and choosing among them).

I also compare the performance of `PointRealEst` and 3Sigma in Fig. 4.12. Fig. 4.12(a) depicts the runtime of each scheduling cycle, including generation of scheduling options, evaluation, formulation of the optimization problem, and execution of the solver. Fig. 4.12(b) reports the runtime of the solver. For both systems, the solver execution is a non-trivial fraction of the scheduling cycle runtime. I observe that distribution-based scheduling also results in a moderate increase in worst-case solver time. As noted in §4.3.3, distribution-based scheduling induces a moderate increase in the number of constraint terms but does not change the number of decision variables. Also note that the actual impact on the solver runtime is upper-bounded by a solver timeout parameter, so the impact of solving on scheduling latency is bounded.

4.5 Summary

3Sigma's use of distributions instead of point estimates allows it to exploit job runtime history robustly. Experiments with trace-derived workloads both on a real 256-node cluster and in



(a) Scheduling Cycle (b) Solver

Figure 4.12: 3Sigma scalability as a function of job submission per hour. Cluster: GOOGLE, Workload: SCALABILITY-*n* where $n \in [2000, 3000, 4000]$

simulation demonstrate that 3Sigma's distribution-based scheduling greatly outperforms a state-of-the-art point-estimate scheduler, approaching the performance of a hypothetical scheduler operating with perfect runtime estimates.

Chapter 5

DistSched: a resource-runtime distribution based scheduler

Applications running on modern computing clusters have highly diverse resource requirements. For example, data analytics jobs are CPU-intensive while distributed in-memory databases are memory-intensive, and others may be constrained on both resources at the same time. In addition, many cluster operators collocate tasks on a machine to improve cluster utilization and throughput. Consequently, cluster schedulers face the challenge of packing tasks onto machines while being respectful of all dimensions of resources available in the machine.

Recent schedulers exploit knowledge of job resource usage to resolve this challenge. Using such knowledge, schedulers are able to pack tasks more tightly onto the same machine and increase cluster utilization [49, 101], to satisfy the operator's desire to maximize the return over investment. Having accurate information about the resource usage also allows schedulers to control performance fluctuation and meet service-level objectives (SLO) of the business-critical jobs [33, 54].

Most systems require a user to specify the resource requirements (e.g., CPU and memory) of the job at submission time. Usually, estimates are an educated guess from the user or predicted based on the relevant history based on previous resource usage observed for similar jobs. However, analyses of the well-studied Google cluster trace [72] reveal that most user estimates are inaccurate, as they have incentives to deliberately provide over-estimates to prevent adverse effects of the under-prediction. A predictor that relies on the job history, such as 3σPredict, shows better prediction accuracy, but the estimates are not always perfect.

Some cluster administrators rely on heuristics, such as over-committing resources, allocating tasks beyond the capacity of the machines [8, 10], to deal with inaccurate estimates. Both over-estimates and under-estimates are undesirable, as over-estimates naturally lead to under-utilization of the cluster and under-estimates can cause resource contention affecting the performance of all tasks running in the machine. However, often times, heuristics are only a partial solution; e.g., too much over-commitment leads to resource contention.

This chapter describes DistSched, a scheduling system that leverages the distributions of resource usage derived from the history of the jobs run in the past to cope with resource usage uncertainty. It explores whether the lessons learned from 3Sigma (Chapter 4) can be applied to the problem of imperfect estimates due to resource usage uncertainty. I find that the distribution-based scheduling ideas from the previous chapter, such as leveraging the distribution, taking advantage of the much richer information, and mis-estimate mitigation mechanisms to make more robust decisions, translate well to the case of resource usage uncertainty. By using more detailed information about resource usage, DistSched can make more robust scheduling decisions that cope well with resource usage uncertainty.

This chapter describes how I apply the distribution-based scheduling approach to a greedy scheduling algorithm, since our analyses show complexities arising from the resource usage uncertainty renders my earlier MILP optimization-based scheduling approach intractable. The environment that DistSched operates in, where more than one task can be scheduled in a machine and resource contention due to resource usage uncertainty is possible, changes the structure of the optimization problem significantly and inflates the number of variables exponentially. Some of the mechanisms for tackling the scalability challenges in 3Sigma did not translate to the new scheduling problem.

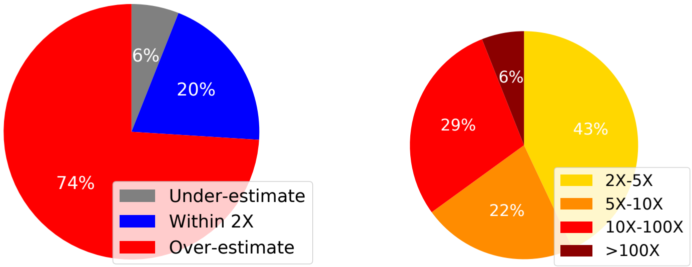
Simulation experiments driven by the Google trace confirm the scheduler's effectiveness. DistSched performs nearly as well as the hypothetical system with perfect estimates, even though distribution estimates from the history-based predictor are imperfect. The system outperforms state-of-the-art point-estimate-based schedulers that use the point prediction derived from the history or from user-provided estimates. The system provides higher SLO attainment for deadline-oriented SLO jobs and increased overall work completed by the system.

This chapter makes four primary contributions. First, it identifies a major problem with recent systems that leverage user-provided resource usage predictions: the majority of the estimates are terrible estimates, and a significant portion of them have very large over-estimate errors. Second, it describes a scheduling algorithm that leverages distribution estimates derived from the relevant part of the job history while reflecting the risk of contention arising from the placement to solve this problem. Third, it describes new scheduler mechanisms to mitigate the effects of outliers that deviate from the observed history. Fourth, it reports on end-to-end experiments on a simulated 700-node cluster, showing the system's effectiveness in robustly exploiting runtime distributions and mitigation techniques to improve SLO attainment and goodput.

5.1 Background

As noted in Sec. 2.2, accurate knowledge of resource usage can be exploited to significant benefit at schedule-time. Such knowledge allows schedulers to pack tasks more tightly onto machines, increasing the cluster utilization [49, 101], and controling performance variation to meet their performance service-level agreements [33]. Some systems use the information to enforce fairness [36, 38] in resource allocation across multiple resource types. This leads to more efficient use of cluster resources, higher service-level objectives attainment for business-critical production jobs with completion deadlines and reduced latency for latency-sensitive best-effort jobs [24, 33, 54, 88].

Thus, many recent systems [14, 38, 40, 54] make use of resource requirement estimates provided by users or predicted from previous runs of similar jobs. However, the effectiveness of such systems rely on the accuracy of the information; otherwise, they may face severe performance penalties.



(a) All Jobs (b) Over-estimates

Figure 5.1: Compares the user provided CPU request amount to the actual average CPU usage of the jobs in the Google trace. (a) Only 20% of jobs are within 2X error range and 74% of jobs are over-estimated. (b) Further examination of over-estimates shows that a significant portion is significantly over-estimated.

5.1.1 Predictability

Analysis of resource usage estimates suggests that, to obtain more accurate runtime prediction, the predictor needs to reflect the placement of the task and its impact on the resource usage rather than just relying on a priori information from the history. This section discusses our analyses of the Google trace. I report four primary observations:

First, only 20% of the jobs are submitted with a CPU-need etimate that is within 2X error range and the majority (74%) of the jobs were provided with an over-estimate (Fig. 5.1(a)). Further breakdown (Fig. 5.1(b)) of the jobs with an over-estimate shows that most of the over-estimates are very large over-estimates. I believe this is a direct consequence of a popular mitigation technique, *padding the estimates*, as the under-prediction results in dire consequences, e.g. slowdowns and higher chance of eviction, rather than low predictability of the resource usage.

Second, the resource usage for most jobs can be accurately predicted from the history of jobs run in the past. The error profile of the resource usage estimates (CPU, memory, CPU-time, and memory-time) from the modified version of 3σPredict ("the predictor") show that most resource usage estimates except memory-time are more predictable (more fraction of jobs lies within 2X error range) than the runtime accuracy (Fig. 5.2). Low predictability of memory-time, despite high predictability of memory, suggests that job's runtime is mostly dependent on CPU-time and CPU, in other words, the task will run as long as it needs to finish the work (CPU-time) for a given amount of compute (CPU) allocated regardless of the memory it needs.

Third, lengthy historical data is not needed to achieve a good level of accuracy. To illustrate this point, I compare the error profile of the estimates for the jobs submitted in week 1 to week 4, starting from zero data in the database and gradually updating the database as the jobs finish in the trace in Fig. 5.3. I note that, for all values, the error profile quickly converges to similar levels after week 1.

Fourth, more interestingly, CPU-time and memory are much more predictable than the runtime or CPU. This suggests CPU-time and memory more consistent across different instances of similar jobs regardless of the placement, while CPU and runtime are more affected by the particular allocation made by the scheduler for each instance.

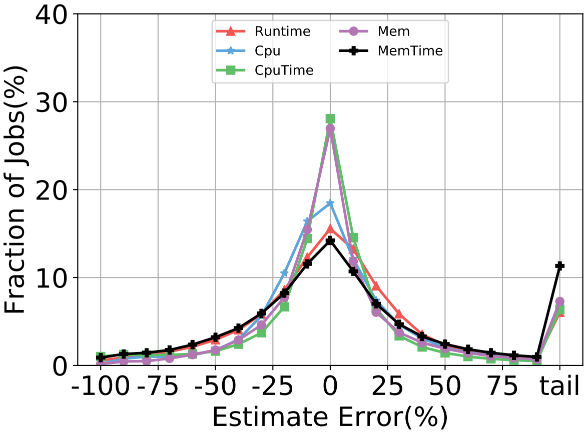


Figure 5.2: Histogram of Precent Estimate Errors comparing estimates from the 3σPredict modified to provide estimates for CPU, Memory, CPU-time, Memory-time. Estimate Error values computed by $\frac{\text{estimate}-\text{actual}}{\text{actual}}$. Each datapoint is a bucket representing values within 5% of the nearest decile. The ”tail” datapoint inclues all estimate errors $> 95\%$.

To confirm this, I ran the following experiment. Fig. 5.4 compares the performance of 3σPredict to the runtime predictors derived from the CPU-time. More precisely, these derived predictors produce a runtime estimate by dividing the CPU-time estimate by (1) the CPU estimate (`PredictedCpu`), (2) the actual average CPU (`ActualCpu`), (3) the CPU request from the user (`Requested`), and (4) the initial CPU usage of the task (`InitialActualCpu`). I observe that `ActualCpu`, a predictor that reflects the actual CPU usage, which is a quantity that reflects its placement, gives the best runtime estimates, outperforming all other predictors including 3σPredict.

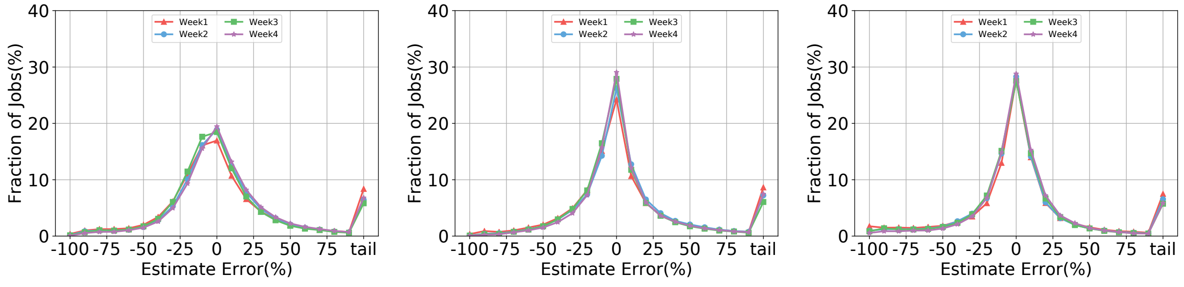
Our conclusion is that, while quantities like CPU-time and memory can be safely predicted by the history-based predictors, the CPU usage and the runtime is actually a function of job characteristics AND the allocation. In order to get an accurate estimate, the predictor must also consider the placement and its impact on the resource usage, rather than attempting to predict purely using a priori information from the history.

5.1.2 Mitigation strategies

The research community has explored techniques to mitigate the effects of mis-estimates, which can significantly hamper a scheduler’s performance.

Intentional over-estimation, padding an additional buffer to an estimate, is frequently used to reduce under-estimation. Under-estimates are problematic, as they trigger resource contention when jobs are packed tightly to maximize cluster utilization. Resource contention adversely affects the performance of running jobs and may cause an SLO violation. Under-estimated jobs are considered to be primary targets of eviction in some systems [14].

However, over-estimation naturally leads to under-utilization of the cluster. Cluster schedulers can tackle this by over-committing resources [8, 10], allocating tasks beyond the capacity of the machine. However, over-committing too much can also cause resource contention.



(a) CPU (b) Memory (c) CPU-time

Figure 5.3: Compares the estimate accuracy of jobs in first week and last week of the Google trace for Runtime, CPU, Memory, and CPU-time. Estimate accuracy is improved over time for all metrics, except for CPU-time.

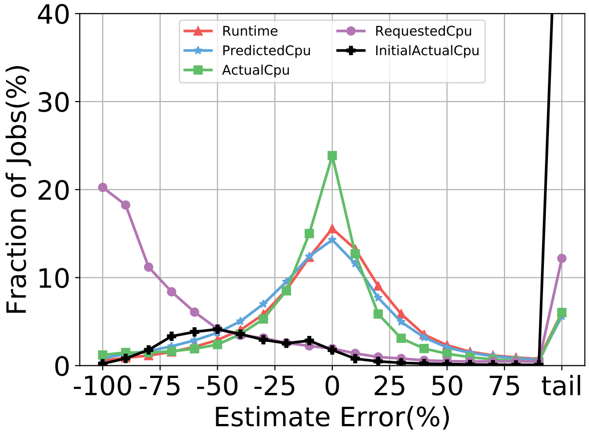


Figure 5.4: Histogram of Percent Estimate Errors comparing runtime estimates computed from different predictors. ”Runtime” estimates are the runtime estimate from 3σPredict, while other estimates are computed by dividing CPU-time estimate from 3σPredict by the respective CPU values.

Morpheus [54] re-assigns resources to jobs that require more resources at runtime. Note that not all applications are designed to be *elastic* and able to make use of additional resources.

Preemption and migration can be applied to address some issues arising from mis-estimates. It can be used to re-assign resources to new high-priority jobs, either by killing or migrating jobs.

These heuristics help to some degree and DistSched makes use of some of them, but these by itself do not eliminate the problem.

5.1.3 Assumptions

This chapter makes a number of assumptions about the cluster, the workload, and resource contention behavior. Some of the assumptions are derived from the behavior of existing operating systems (e.g., Linux) and cluster management systems (e.g., Hadoop YARN [89], and Kubernetes [14]), and some are caused by the limited information available in the Google cluster trace on which DistSched is evaluated.

Each job consists of one or more tasks, where each task represents a Linux program to be run on a single machine. Each task is accompanied by its resource requirement, such as how much CPU (in terms of fractions of a core) and memory is needed. During the course of execution, the task will consume a varying amount of CPU cycles and memory. The task runtime is determined by the amount of work in terms of CPU-time and amount of CPU-time it is allocated for. In other words, a task will finish execution if it acquires enough CPU-time to complete its amount of work.

Users submit jobs to the cluster scheduler that manages a collection of machines in the cluster connected by a high-bandwidth network. This chapter assumes all machines are *homogenous* apart from the amount of resource available in the machine, e.g., a task's runtime given X% of a core will be the same on any machines in the cluster.

This chapter considers only two types of resources available to be used by jobs: CPU and memory. CPU and memory are distinguished by the system's behavior when the sets of tasks executing in a machine attempt to use more resources than available in the machine: CPU resource is time-shared while memory is considered to be inflexible as most systems consider *thrashing* a failure condition.

Contention for CPU resources, when running tasks attempt to use more CPU than is available, is considered to be a *soft* failure. The OS task scheduler (Completely Fair Scheduler for Linux) will time-share available CPU in the machine. It allocates available CPU cycles to tasks according to a proportionate share of their resource request (# of cores of CPU * CPU request of a task / aggregate task request of all tasks in the machine).

Memory contention is considered to be a *hard* failure for most systems. The cluster scheduler evicts running tasks (in the order of importance and amount of resource over-usage) as necessary, if the aggregate memory consumption exceeds the pre-defined threshold. The threshold is set to be less than the actual machine capacity to preclude invocation of the OS level Out-of-Memory manager, which may evict running processes in an undesirable order, including crucial processes running in individual worker machines on behalf of the cluster scheduler.

Other types of resources, such as IO (memory and disk) or network bandwidth are not considered in this chapter due to limitations of the information available in the trace. However, I expect bandwidth resources to be time-shared similar to CPU resources. This chapter also does not consider cache hierarchy, NUMA-ness, or the network topology.

5.2 Resource distribution-based scheduling

This section discusses mechanisms that enable schedulers to leverage resource distributions, as opposed to point estimates.

The runtime behavior of the task, its resource usage and runtime, will depend on the allocation decisions. However, a fundamental departure from the previous chapter is that not only the static properties of the machine (such as "GPU is present"), but the resource usage of the other tasks currently running in the machine also affects the performance of the task being scheduled.

Nevertheless, the general framework for the scheduling algorithm is similar. The scheduler produces all possible placement options, associates each with an expected utility value according to the distribution, then chooses to run the set of jobs that both maximize the overall sum of utility and fit within the available resources. This section describes how the expected utility is calculated and the mechanism to choose a set of jobs to schedule.

5.2.1 Valuation of scheduling options

For each incoming job, there will be a set of possible scheduling options. The placement of the job dictates the completion time and its usefulness or utility. The scheduler needs to place the job in a way that maximizes the overall utility. This section describes how to associate a utility value to a scheduling option of the job.

Utility. The scheduler uses utility functions to map job completion times to the utilities of job placements. The utility function of the job will be provided by a cluster administrator or an expert user, such as the utility functions defined for SLO and latency sensitive jobs in Chapter 4.

Runtime. The runtime of the job is defined as the maximal runtime of its tasks.

The runtime of each task is computed as the CPU-time (the amount of work), divided by the actual CPU usage of the task. The actual CPU usage can be different from the task's intrinsic CPU (CPU usage if there isn't any resource contention). If there is a contention, i.e. when the aggregate CPU demand of the running tasks is greater than the available in the machine, the process scheduler will allocate cycles according to the share (CPU request of the task divided by the total CPU request in the machine) of the running tasks.

Thus, the CPU usage of the pending task is calculated as follows:

1. If the intrinsic CPU is smaller than or equal to the task's share of the machine CPU (machine CPU capacity × CPU share of the task), the CPU usage will be the intrinsic CPU.
2. If the usage of the running tasks in the machine is more than their share of the machine CPU, the CPU usage will be task's share of the machine CPU.
3. Otherwise, the task will be able to use idle CPU capped at the intrinsic CPU of the task.

Probability of the over-allocation. The job will not reach its completion and attain any utility if any of its tasks is evicted during its execution. Thus, the scheduler needs to calculate the probability of the task eviction, the probability that aggregate memory usage of the task and the running tasks will exceed the machine capacity. Assuming the intrinsic memory usage of the tasks is independent, the probability of a task eviction can be calculated as the probability that allocation will result in the memory overuse:

$$P(\text{task eviction}) = \sum I(n \times \text{Mem}_{\text{task}} + \text{Mem}_{\text{running}} > \text{Machine capacity}) P(\text{Mem}_{\text{task}}) P(\text{Mem}_{\text{running}}) \tag{5.1}$$

where n is the number of tasks allocated to the machine, $\text{Mem}_{\text{running}}$ is a random variable representing aggregate memory usage of the running task on the machine, and I is an indicator function.

Assuming task eviction events are independent, the probability of not having any of the task for a job evicted is $P(\text{no-eviction}) = \prod_m 1 - P(\text{task eviction}_m)$.

Aggregate resource usage. Computing the aggregate resource usage of the running tasks on a machine can be tricky. Even though current usage levels are observable, there is still a risk that the resource usage will change in the future. Therefore, the resource usage of the running task is also assumed to be an uncertain value that follows the distribution. The aggregate of resource usages will also be an uncertain value that follows its own distribution.

If I assume the distribution of resource usage for different jobs are independent, the distribution of aggregate resource capacity can be calculated as if I am adding independent random variables. For example, if there are two tasks with uncertain resource usage RC_1 and RC_2 running in the machine, the probability of the aggregate usage being x is the probability that addition of RC_1 and RC_2 yields x:

$$P(\text{Aggregate} = x) = \sum_0^\infty P(\mathrm{RC}_1 = k)P(\mathrm{RC}_2 = x - k)dk \tag{5.2}$$

Expected Utility. For each placement option, the scheduler computes the expected utility of a job using the distributions of resource usage. The expected utility is calculated as the sum of the utilities for each possible runtime T, weighted by the probability that the job runs for T and the probability that no task will be evicted.

$$E[\text{Utility}] = P(\text{no-eviction})E[U(T)] \tag{5.3}$$

To compute the second term of the expression, I note that the job runtime is the runtime of its longest task and each of task runtimes is dependent on the random variables, the task's intrinsic CPU usage and the aggregate intrinsic CPU usage for the running task on the machine. Thus, the term needs to be calculated by enumerating all possible combinations of the task's intrinsic CPU usage, and the aggregate intrinsic CPU usage of the running tasks:

$$E[U(T)] = \sum_{\text{Cpu},\text{Cpu}_{m1},\ldots,\text{Cpu}_{mn}} \max\left(\frac{\text{Work}}{\text{Actual Cpu}_i}\right)P(\text{Cpu})P(\text{Cpu}_{m1}),\ldots,P(\text{Cpu}_{mn})$$

where the actual CPU usage of the task is computed as described previously and the work is the point prediction of the CPU-time values of the task.

Although the estimate of the work can also be considered as a variable quantity with its distribution, above uses a point prediction to simplify the calculation instead of enumerating on possible CPU-time value as well and maintaining joint distributions of CPU usage and CPU-time, as two quantities are obviously correlated to each other and cannot be separated. I believe this is safe as CPU-time estimates and actual CPU usage have shown high estimate accuracy compared to runtime or CPU usage.

5.2.2 Scheduling challenges

Given the evaluation of the scheduling options, the scheduler needs to choose the optimal combination of scheduling options to maximize the overall expected utility. The high-level intuition is to bin-pack jobs, each represented as a three-dimensional resource-time rectangle, into a cluster resource-time, where one dimension represents time and the other two dimensions represent Cpu and memory respectively. One approach is to cast the problem as a classical optimization problem, in the form of Mixed Integer Linear Programming (MILP).

This section describes the MILP representation for the simplest case, where resource usage of all tasks are precisely known in advance and discuss why the formulation becomes intractable if I consider a resource usage as an uncertain quantity that follows some distribution. Our conclusion is that the MILP representation for the scheduler that can consider resource distribution and cope with resource contention is significantly more complex compared to previous systems (Chapter 4, [88]), and the scheduler must resort to a greedy algorithm in order to make scheduling decisions in an acceptable amount of time.

Precise estimate scenario. I first describe the scheduling problem for the simplest case where resource usage is precisely known.

The scheduler starts by enumerating all possible scheduling options for each job. A scheduling option o for a particular job j is a function that maps each machine m in the cluster to the number of tasks of the job j allocated to the machine. Each placement option o for a given job j is associated with a binary decision variable I_{jo} and the number of tasks for job j allocated to machine m is represented by an integer-valued machine allocation variable M_{jm}. A constraint $kI_{jo} = \sum_m M_{jm}$ binds the decision variable I_{jo} and machine allocation variable M_{jm} to ensure all k pending tasks are allocated when the job j is chosen.

For each scheduling option for job j and option o, a constant utility value U_{jo} calculated by evaluating the utility curve (Fig. 4.3) at the expected completion time of the job. The objective of the scheduling problem is to find the combination of scheduling options such that the overall expected utility of options is maximized. Then, the objective function to maximize is the sum of objective values modulated with respective decision variable: $\sum_j \sum_o U_{jo} I_{jo}$. Constraints are introduced to ensure only one scheduling option is selected for each job ($\forall j \sum_o I_{jo} \leq 1$).

The optimization problem also needs to ensure the tasks are allocated with each machine's resource limits. The scheduler auto-generates a set of constraints that provide invariant that

$$\forall m \sum_j \mathrm{Cpu}_j M_{jm} \leq \mathrm{Cpu}_{\text{machine capacity}} - \mathrm{Cpu}^m_{\text{running}} \tag{5.4}$$

$$\forall m \sum_j \mathrm{Mem}_j M_{jm} \leq \mathrm{Mem}_{\text{machine capacity}} - \mathrm{Mem}^m_{\text{running}} \tag{5.5}$$

where Cpu_j is the task CPU usage of job j and $\mathrm{Cpu}^m_{\text{running}}$ is the aggregate CPU usage of running tasks in machine m. This provides the desired guarantee that no tasks are allocated beyond the capacity of a machine.

The solution that maximizes this problem will effectively select which jobs to run and which placement option among the options for the picked job needs to be chosen, while making sure no machine is over-allocated.

Discussion. Even for the simplest case, the number of available scheduling options for a job is the number of possible combinations of machines in the cluster. Previous works, 3Sigma (Chapter 4) and TetriSched [88], avoided such explosion by only considering equivalent sets of machines, sets of resources equivalent from the perspective of a given job, such as all nodes with a GPU. However, the notion of an equivalent set does not improve the complexity of the scheduling problem where more than one task can be scheduled on the same machine. In this case, the machines are only equivalent from the perspective of a given task if an identical amount of resources are available. Due to the diversity of task sizes in real workloads, the number of equivalent sets quickly spans all possible pairs of CPU and memory, even after values are discretized to a reasonable number of buckets (e.g., 20 buckets for each resource yields 400 possible pairs).

The problem only gets worse if resource usage is assumed to be not precisely known and resource contention is possible. In this setting, machines are equivalent from the perspective of a task, only if a placement will yield an identical task execution, in terms of resource usage and the runtime. If the resource usage is a variable quantity, the equivalence sets need to be defined by the distribution of aggregate resource usage of all running tasks in the machine, meaning machines within the same equivalent set must be running the same set of tasks (in terms of their resource usage). Otherwise, the resource usage and the rate of progress of the task under contention will be different.

The more fundamental problem is incorporating the task malleability into the MILP formulation. In the current setting, the runtime of the task is not fixed but determined by the actual amount of allocated CPU of the task, which is affected by the amount of CPU that will be used by the other tasks being scheduled to the same machine, as well as the amount of CPU used by the tasks already running in the machine.

First implication is that the scheduler must consider all possible combinations of scheduling options for each job separately, e.g. a combination of scheduling option $o_{1,1}$ of the job j_1 and $o_{2,1}$ of the job j_2, a combination of scheduling option $o_{1,2}$ of the job j_1 and $o_{2,1}$ of the job j_2, and so on. This is because a placement option can yield different expected utility value depending on the placements of the other jobs, as they affect the runtime behavior of the job. To incorporate this in MILP, I must introduce a decision variable for each combination of the possible scheduling options for each job and evaluate aggregate expected utility for each combination separately.

Furthermore, the impact of the task placement on the performance of the running jobs must be incorporated as well. Similar to how a placement option affects the performance of the other task being scheduled, a placement may have an impact on the performance of the running tasks. For example, scheduling one additional task onto the machine may create CPU contention, which may slow down a task running on the machine and ultimately cause it to miss the deadline. Worse, a new task may create memory contention and trigger eviction of an already running task in the machine. This implies the aggregate expected utility for each combination of scheduling options must consider the change in the expected utility of the running jobs as well.

In conclusion, unlike previous work (3Sigma Chapter 4 and TetriSched [88]), I believe the MILP formulation for the scheduling problem that DistSched deals with, where multiple tasks can be placed onto the machine especially when the resource usage of the task is uncertain and can cause resource contention, is more difficult compared to the one faced by 3Sigma. This is also demonstrated through a preliminary evaluation using the MILP formulation of the scheduling problem, which fails to solve the problem in reasonable amount of time (minutes) for a realistic cluster size and number of jobs in the system. Thus, we conclude that additional innovations are needed to address the scheduling problem in a scalable way. Therefore, DistSched relies on a greedy scheduling algorithm that is similar to the algorithms used by the schedulers deployed in practice.

5.2.3 Greedy scheduling algorithm

This section describes the greedy scheduling algorithm used by the scheduler, which places newly arriving jobs and their tasks onto the available machines in the cluster. The discussion includes how the scheduler constructs a feasible placement option for each job and chooses a set of jobs to schedule in a greedy manner. I conclude the section by examining scalability issues arising from the complexity of the algorithm.

Setup. Scheduler operates on a periodic cycle, making placement decisions for all pending jobs in the queue in an online scheduling setting. The primary objective of the scheduler is to maximize the overall utility of the jobs being scheduled.

The inputs to the scheduler are:

(1) Queue of pending job requests: Each request contains the number of tasks being requested, the amount of resource requested by the user, the estimated task resource distribution (CPU and memory), the estimated distribution for the amount of work for each task, the job's priority, and a deadline if it is an SLO job.

(2) Status of the cluster: The scheduler also keeps track of the running jobs in the cluster, the amount of work completed so far by each task, and the amount of resource currently being used.

To make the scheduling problem tractable, I elect the following simplifications:

1. The algorithm will select a feasible placement option for a job by greedily choosing a machine to schedule a task, instead of considering all possible combination of machine placement available regardless of its feasibility.
2. The scheduler will construct the list of jobs to schedule, one job at a time, instead of considering all possible combination of jobs to schedule.

The scheduler will repeat the selection process with an updated state of the cluster until no job can be scheduled onto the cluster. The jobs that are not yet schedulable in the cycle will be considered in the next scheduling cycle.

Construction of scheduling option. At the beginning of the scheduling cycle, the scheduler first constructs a feasible placement option for each job in the pending queue.

For each job, the scheduler will consider machines in the cluster where a task allocation is feasible. A machine is considered to be feasible if the probability of resource over-allocation is less than the user-specified threshold. This practice of *optimistic over-packing* will prevent the scheduler from being too conservative in the task placement because of infrequent outliers observed in the past.

The scheduler creates a sorted list of the feasible machines in the cluster in ascending order of the following metrics: (1) probability of memory over-allocation, (2) probability of CPU over-allocation, (3) amount of available memory, (4) amount of available CPU, (5) a random number to break the tie. The probabilities of resource over-allocation are considered first, as over-allocation has a detrimental effect on the runtime of the pending job, as well as the performance of the already running jobs. Given a similar risk of resource over-allocation, the scheduler would prefer machines that have a smaller amount of resource available. This will have the desired effect of packing task onto machines that already occupied first while leaving more machines idle to accommodate larger tasks later on. Finally, a random number is introduced so that machines will be selected randomly rather than in a particular order.

Given a list of a machine to choose, the scheduler will greedily select the machine at the top of the list to schedule a task. After a machine is chosen, the scheduler simulates a task allocation by adding the task's resource usage distribution (using Eq. (5.2)) to the aggregate resource usage distribution and decrementing the expected resource usage to the available resource of the machine. If the machine is still feasible after the allocation, the machine is placed back on the sorted list according to the updated metrics after allocation. This will allow the scheduler to schedule more than one task onto the same machine. The scheduler repeats this process until enough machines have been chosen to schedule the job or if no feasible machine is available. The job is considered to be schedulable if the scheduler was able to find enough feasible machines to schedule all of its tasks.

Selection. Given a list of schedulable jobs, the scheduler will choose the job with the highest expected utility (Eq. (5.3)) to schedule. The scheduler makes note of the decision, updates the status of the cluster accordingly, and repeats the process of scheduling option construction and job selection until no job is schedulable.

Scalability. Even after making the above simplifications, there are some parts of the scheduling algorithm that are computationally expensive. I apply a number of optimizations. First, the scheduler maintains the distribution of the aggregate resource usage for each machine and recomputes only if it is necessary (e.g., when a new task is started or a running task is finished). The re-computation is expensive, because the scheduler needs to enumerate all possible combinations of resource usage for each of the task running in the machine in order to compute the aggregate distribution. Even if a discrete distribution with a limited number of possible outcome is assumed (as in Chapter 4), the runtime is exponential to the number of tasks running in the machine. Second, the scheduler prunes possible values of the aggregate resource usage when computing the expected utility. Instead of enumerating all possible outcomes, the scheduler will consider outcomes that will lead to over-allocation and merge outcomes that do not over-allocate as a single outcome during computation. In tandem with the user-specified resource over-allocation threshold, this optimization bounds the number of combinations to consider to a tractable number.

5.3 Implementation

This section describes the architecture of DistSched. The system will replace the scheduling component of a cluster manager (e.g. YARN or Kubernetes). The cluster manager will remain as an orchestrator that is responsible for the life cycle of the cluster.

The user submits the job request to the cluster manager. The requests include a number of attributes, such as the name of the job, the user submitting the job, and the specification of the resources requested. The cluster manager forwards the request to `DSPredict`.

The role of `DSPredict` (Sec. 5.3.1) is to provide an estimate of a probability distribution of the resource usage of the submitted job. `DSPredict` achieves this by maintaining a history of previously executed jobs, identifying several candidate groups of similar jobs, and constructing the distributions for the submitted job. `DSPredict` forwards the predicted distributions along with the job request to `DSSched`.

`DSSched` makes the scheduling decision using the distribution of resource usage and request specifications. The scheduler is invoked in regular interval (every 15 seconds) to consider all jobs that were not scheduled or arrived after the previous invocation. `DSSched` comes up with a possible placement option for the job and computes the expected utility of the job (Sec. 5.2.3). It will decide which jobs should run first and where to schedule the tasks of the job and notify the cluster manager to actually start the job.

The cluster manager is responsible for the actual execution of the job in the cluster. The system expects the operating system on the machines is running its own CPU scheduler (e.g. CFS scheduler for Linux) and out-of-memory (OOM) killer in case the resource demand exceeds the resource available on the machine.

At the end of the job's execution, the job's actual runtime is passed to `DSPredict` and incorporated into the job history for future predictions.

This section details how `DSPredict` estimates resource usage distributions and how the scheduler handles mis-estimation of the resource usage.

5.3.1 `DSPredict`

`DSPredict` is the component that provides distribution for resource usage, namely CPU, memory, and CPU-time. There has been a plethora of previous literature on the history-based predictor of resource usage, and this dissertation does not attempt to innovate on that front.

Instead, I use a modified version of 3σPredict (Chapter 4) to produce the distribution estimates of CPU and memory usage, as well as the amount of work, CPU-time. To come up with the estimate of an incoming job, 3σPredict first identifies the several candidate groups of similar jobs observed from the history (e.g., jobs submitted by same user, jobs with similar job name, etc.), chooses the group that has shown the lowest point prediction error, and sends the empirical distribution constructed from the group to the scheduler as the distribution estimate of the job.

`DSPredict` does not make any assumption about the shape of the distributions. The system uses an empirical distribution, represented as a histogram of numeric values (e.g. CPU, memory, CPU-time). The predictor constructs and maintains the distribution of each possible candidate group using a streaming histogram algorithm [11]. This allows the predictor to maintain an approximate empirical distribution of each candidate groups in streaming fashion without needing to iterate through the job history every time.

5.3.2 Mis-estimate mitigation strategies

`DSPredict` provides the distribution of a job's resource usage using the history of previously executed jobs. However, the distribution may not be perfect in practice. It is possible for the job to have an insufficient history or the distribution to evolve over time (e.g., code changes, different input data). The scheduler uses the following mitigation techniques.

Runtime over-estimate handling. While `DSSched` does not use the distribution of the runtime directly, there are still cases where `DSSched` may believe the job is impossible to finish on time. In DistSched, the task completion time is considered to be the predicted amount of CPU-time divided by the CPU usage. If CPU-time is over-estimated or CPU usage is under-estimated, possible task completion times can be greater than the time to deadline. In this case, the expected utility is zero, and the scheduler will not see any benefit from spending resources on the job even if there are idle resources available. To mitigate the effect, the scheduler employs the over-estimate handling technique from the previous chapter, wherein the scheduler uses a utility function with a linearly decaying slope past the deadline (Fig. 4.3(d)). This allows the job to have non-zero utility while being lower than other SLO jobs submitted with the same initial utility, allowing the scheduler to schedule seemingly impossible jobs only if there are idle resources.

CPU-time distribution under-estimate handling. For restarted tasks (e.g., after preemption or eviction), the scheduler updates the CPU-time distribution, as it has additional information, namely the amount of work completed in its previous execution, which is a lower bound of the actual CPU-time that needs to be completed by the task. The scheduler updates the distribution by using a conditional probability density $P(\text{CPU-time}|\text{CPU-time} > \textit{elapsed})$, rather than the original distribution for the CPU-time. This is especially useful, when the distribution is multi-modal as this process will quickly ignore the part of the history that is irrelevant. When the actual amount of work exceeds all historical data points for the CPU-time distribution, the scheduler uses the amount of work completed by the task so far as an indication of the remaining amount of work for the task, which is the best estimate of remaining work absent any other information [42].

Distribution from the observed resource usage. During the job's execution, the cluster manager can observe the actual resource usage of the job. For most cases, the resource usage lies within the range of the predicted distribution, but some jobs' resource usage can be drastically different from the prediction. This is especially problematic if the predicted distribution is under-estimated, where the actual usage is constantly above the maximal possible value from the distribution. This may cause resource contention when more jobs are scheduled to the same machine. When the scheduler notices an under-estimated task, it will assume the predicted distribution is inaccurate, and use the distribution of resource usage constructed using the actual resource usage of the task instead from now on. The new distribution will have possible resource usage values ranging from the minimum and maximum resource usage observed in the current execution of the task, where the density of each value will be the proportion of time that the specific usage value has been reported divided by the elapsed time of the job.

Preemption. Even with other mitigation techniques, resource contention may be inevitable. This may happen if the scheduler fails to update the mis-predicted resource usage on time, or the scheduler was simply *unlucky*, e.g. all tasks scheduled on the machine started to consume resources at the higher end of their distributions. If there is a contention in the cluster, the scheduler will preempt lower priority best-effort jobs to free up resources on a contended node in order to allow more important SLO jobs to complete on time. The jobs that use more resources than asked for, use more resources than the predicted resource distribution, create contention in more machines, and use a larger amount of resources will be the primary candidates for the preemption. The scheduler will preempt tasks until the machine utilization falls below the predefined threshold.

5.4 Evaluation

This section evaluates the new scheduling system in a simulated environment, yielding three key takeaways. First, the system outperforms the state-of-the-art system in SLO miss rate and goodput, approaching the performance of the unrealistic `Actual`. Second, the components of the system are all important as demonstrated through a piece-wise benefit attribution. Third, I show that the system can scale even with additional features that add computational overhead.

5.4.1 Experimental setup

I conduct a series of experiments to evaluate the system in a simulated setting.

Simulator. To evaluate the system, I designed a simulator that can read a workload trace and faithfully simulate jobs in the trace in the simulated 700-node cluster.

The workload trace contains the runtime parameters of the jobs, such as arrival time, the number of tasks in a job, deadline, the amount of resource being requested by the user, and job attributes used by `DSPredict` when coming up with estimates of resource usage. The trace also contains a time series of the resource usage (tuples of duration, CPU, and memory) of each task, which represent the transition of resource usage during their execution.

The simulation is in the form of discrete event simulation. The simulator maintains a state of the cluster, machines, and running jobs and their tasks. The state is mutated as the events are simulated in chronological order.

Events consist of job life cycle events, task resource usage change events, and scheduling events. Job life cycle events include job submissions and task terminations. Resource usage change events adjust the resource usage of the running tasks. Regularly-invoked scheduling events invoke the scheduler to consider jobs pending in the queue for scheduling.

The simulator maintains a priority queue of events ordered by timestamps of the events. At the beginning of the simulation, the queue contains job submission events for all jobs from the workload trace, as well as periodic scheduling events (invoked every 15 seconds for the experiments). When a job is submitted, the simulator inserts the job request into the pending job queue. At each scheduler event, the scheduler only considers jobs in the pending job queue. When a job is scheduled, the simulator creates a resource usage change event for each resource usage transitions for each task, as well as the task termination event, and inserts them into the queue. During a resource usage change event, the simulator attempts to adjust the resource usage of the task. If there is not enough idle resource on the machine, a separate event to relieve resource contention is inserted into the event queue (with identical timestamp). The event is simulated after all resource usage change events for the machines are simulated.

The simulator simulates and relieves resource contention in a manner similar to the behavior of Kubenetes [14]. Memory contention is handled using task evictions, as the amount of memory in the system cannot be gracefully shared like CPU cgroups. When the aggregate memory usage of a machine reaches the threshold, the simulator sorts the tasks running in the machine by ascending order of priority (SLO jobs have higher priority over BE jobs), then by descending order of the amount of memory used above the entitled share and by descending order of the total memory used. The simulator evicts the task from the top of the list until the contention is resolved. When a task is evicted, the simulator removes all future resource usage change events associated with the task from the event queue and terminates the task right away.

For CPU contention, the simulator emulates the CPU sharing behavior of Linux cgroups. Each task will be entitled to its allocated share computed by dividing its CPU request by the aggregate CPU request of all task running on the machine. If there are remaining resources on the machine (e.g., a task is not using all of its allocated share), the remaining tasks will split the idle CPU according to the ratio of the requests. This process is repeated until all tasks have enough or no more CPU is available in the machine.

When a task's CPU usage is throttled due to a CPU contention, the simulator adjusts the resource usage time series of the task to account for the slowdown, as well as the corresponding resource usage change events and the task termination event. For example, the series includes a tuple (15s, 4 CPU, x memory), which means the task will use 4 CPUs and x memory for the next 15 seconds; but, throttled to 2 CPUs, the tuple will be modified to (30s, 2 CPU, x memory), meaning the next resource usage transition will take place in 30 seconds rather than 15 seconds.

Accordingly, the simulator modifies the corresponding resource usage change event to fire in 30 seconds.

At the end of the simulation, the simulator outputs a log of all events that occurred during the simulation for post-processing.

Systems compared. In the end-to-end evaluation, I compare four different scheduling systems. `Dist` is our system in which the scheduler is given resource distribution provided by the scheduler equipped with mis-estimate handling mechanisms. Other systems use the state-of-the-art point-estimate scheduling approach. `Actual` is a hypothetical system where the scheduler is given an average actual resource usage for each incoming job. `Point` uses a point prediction of the average resource usage from the predictor. `Request` uses a user-provided estimate of resource usage provided from the original trace.

I also compare `Dist` with an alternative way of generating the point estimates. Instead of using average resource usage as a basis for prediction, `DistMax`, `ActualMax`, and `PointMax` use maximal task resource usage observed. `ActualMax` use the actual maximal resource usage for each incoming job, `DistMax` use the distribution estimate from `DSPredict`, and `PointMax` use the point estimate from the predictor based on the maximal resource usage of the jobs run in the past.

To evaluate the benefits of the each of the mechanisms used by `Dist`, I also compare `Dist` against itself running with different configurations, where each would disable one of the features: (a) distribution-based scheduling (`NoDist`), (b) optimistic over-packing (`NoPack`), (c) runtime over-estimate handling (`NoOver`), (d) using distribution from the observed resource usage (`NoObs`), and (e) preemption (`NoPreempt`).

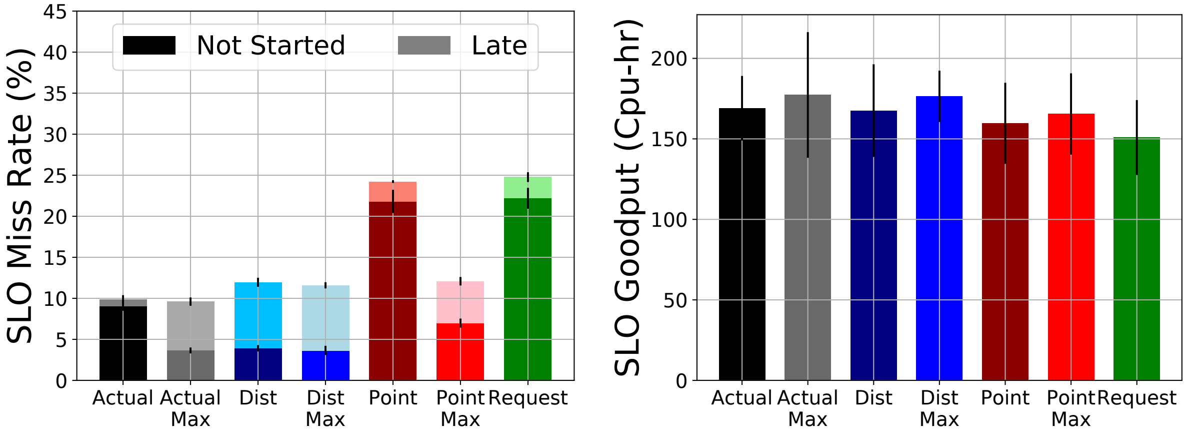
Workload. The experiments use the workload derived from the Google trace [72]. Recall that the amount of resources has been obfuscated to have a value between 0 and 1 in each dimension in the original trace. From the trace, I filtered out jobs that were not successful and randomly selected a segment from the workload that had desired load of 0.8. The selected workload is 2 hours in length and contains approximately 700 jobs.

Workload configuration. As the original trace lacks a deadline for the jobs, I randomly selected jobs, designated them as SLO jobs and assigned generated deadlines. The final workload has approximately even composition of SLO jobs with a deadline and lower priority latency sensitive BE jobs. For SLO jobs, the deadline slack, the amount of time between the arrival time and deadline, is an important consideration. The deadline slack is computed by the time between deadline and arrival time divided by the job's runtime (e.g., deadline slack of 1.2 means a 100s job will have 120s from the job arrival to finish). I select each SLO job's deadline slack randomly from a set of 4 options: 1.2, 1.4, 1.6, 1.8. To measure the experimental variation arising from a random selection of SLO jobs and their deadline slacks, I generate 10 different workloads each with a different selection and repeat each evaluation with these workloads.

Success metrics and metrics of contention. I report the following success metrics for the experiments. The scheduler's primary goal is to minimize the SLO miss rate: the percentage of jobs that miss their deadline. I also measure the total work completed in core-hours (goodput), showing how much aggregate work is completed. Finally, I measure average BE latency – the mean response time for BE jobs.

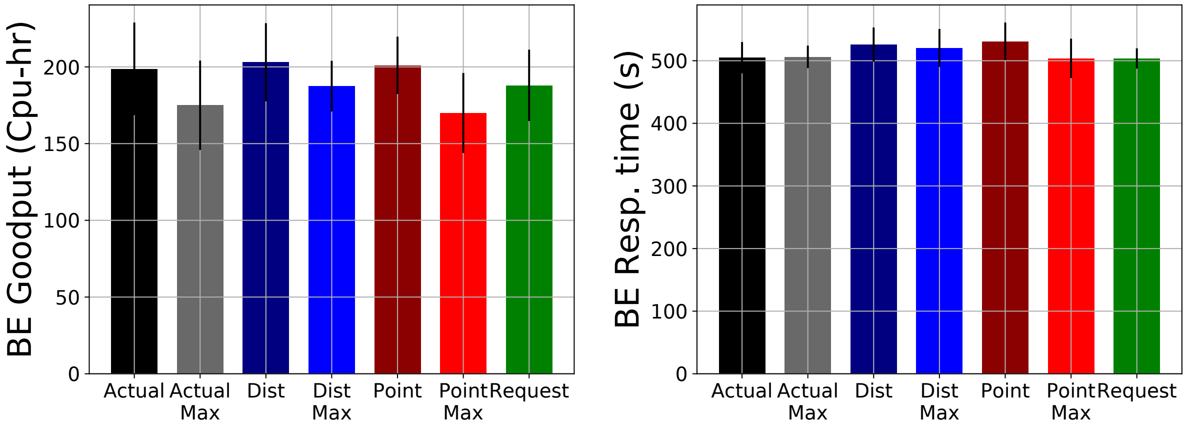
I also report the following metrics to analyze the amount of contention during the experiment. To measure the amount of CPU contention, I report the total number of times the simulator had to throttle CPU usage of the tasks divided by the number of nodes and the experiment duration. As a measure of memory contention, I report the number of tasks evicted because of the memory over-allocation.

5.4.2 End-to-end performance



(a) SLO Miss Rate

(b) SLO Goodput



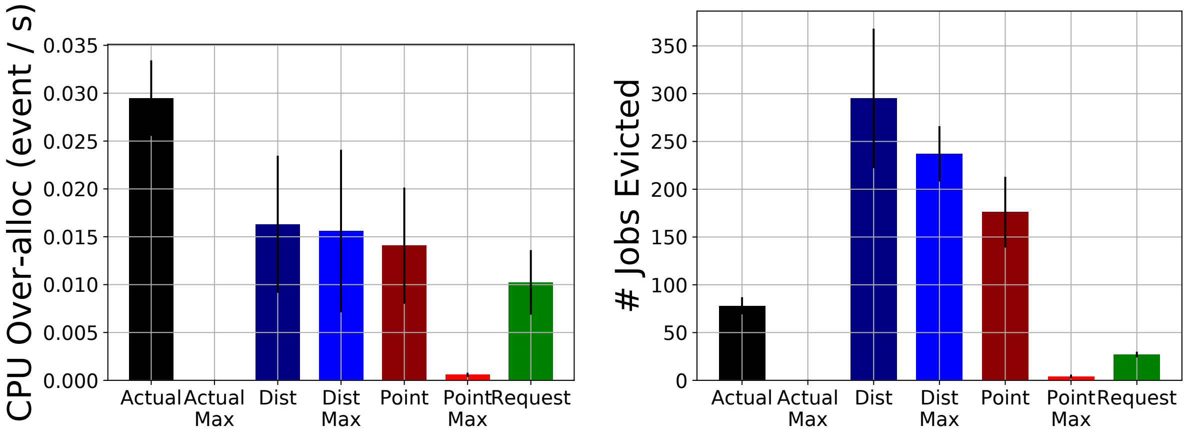
(c) BE Goodput

(d) Avg BE Latency

Figure 5.5: Compares the performance of `Dist` with other systems in the simulated cluster. `Dist` outperforms `Point` on SLO miss-rate and SLO Goodput while nearly matching `Point` and `Request` on BE Goodput and BE Latency.

Figs. 5.5 and 5.6 show performance results for the four scheduling systems, as well as their maximal task resource usage variants.

`Dist` outperforms `Point` and `Request` in minimizing SLO misses, our primary objective, and approaches closely to `Actual` while completing similar or more work compared to `Point` or `Request`. `Dist` performs well even if it does not have knowledge of actual resource usage



(a) Rate of Cpu over-allocation

(b) Job evictions

Figure 5.6: Compares the amount of resource contention as `Dist` and other systems schedule jobs in the cluster. `Dist` outperforms `Point` on SLO miss-rate and SLO Goodput while nearly matching `Point` on BE Goodput and BE Latency.

allowed to `Actual`, as it leverages the resource usage distribution to make an informed decision, such as whether the task will fit into the machine and whether the job will complete on time. However, `Dist` is not perfect. A part of `Dist`'s success stems from its aggressiveness in starting jobs compared to `Actual`. As a result, `Dist` cause more jobs to be evicted due to memory over-allocation (Fig. 5.6(b)). BE latency is similar across all systems compared.

`Point` incurs much higher SLO miss-rate (2.15x-2.2x) and lower SLO goodput compared to `Dist`, as it struggles with realistic prediction error profiles from a history-based predictor. Most of the SLO misses come from not starting enough jobs, implying that `Point` is too conservative in starting jobs. Because `Point` lacks a resource under-estimate handling mechanism, it can neglect to schedule SLO jobs that are not predicted to complete on time, even if cluster resources are available. As a result, `Point` will complete more BE jobs compared to `Actual` or `Dist`.

`Request` reports similar levels of SLO miss-rate to `Point`, but much lower goodput completed for both SLO and BE jobs. `Request` also struggles with realistic error profiles, but coming from the user estimates, which tend to grossly over-estimate the resource usage (Fig. 5.1). For jobs with resource over-estimates, `Request` finds it difficult to find a machine that can fit the seemingly larger task, fails to start jobs on time, causing the job to miss its deadline. Thus, unlike `Point` (which has mostly even composition of under-estimates and over-estimates), `Request` fails to fill the cluster resources with suitable BE jobs and completes less amount of work overall.

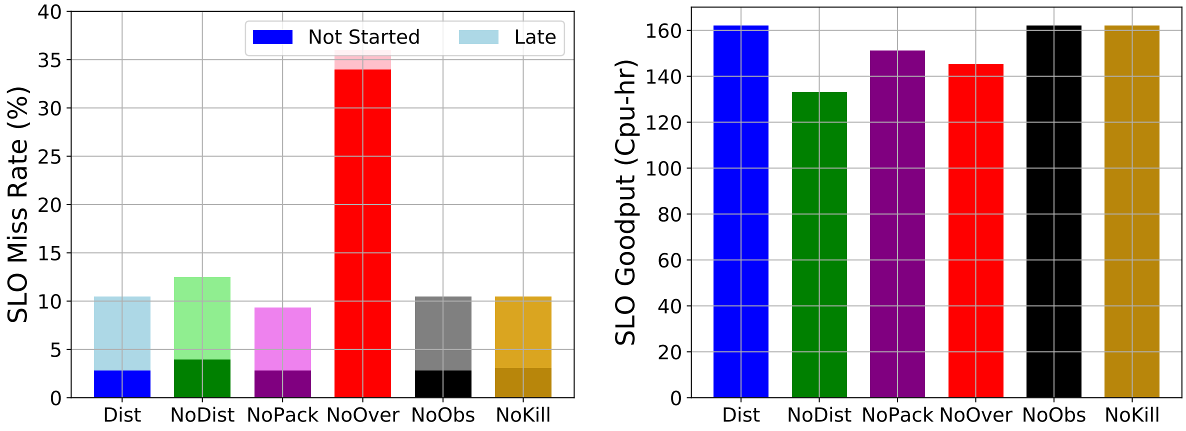
Comparison with max predictors. I also compare the performance of `Dist` with the system variants that run the predictor based on the maximal resource usage of the job.

`ActualMax` improves SLO miss-rate slightly compared to `Actual`, and `PointMax` improves the miss-rate significantly compared to `Point`. Using maximal resource usage estimates naturally inflates the resource usage estimates and leads to a significant reduction in resource contention caused by the resource under-estimates (Fig. 5.6). Thus, `ActualMax` and `PointMax` are able to execute jobs without resource contention, which leads to higher performance.

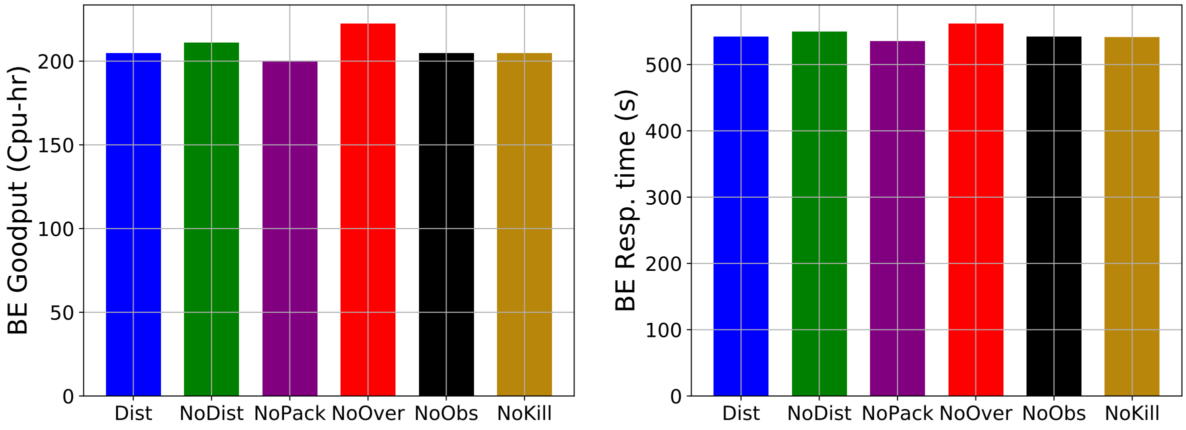
However, using maximal resource usage is not always an optimal solution. Namely, `PointMax` still completes fewer best effort workload compared to `Dist`. This is because some jobs will have a very large over-estimate error, especially if predicted with maximal resource usage. `PointMax` find it difficult to find a machine that can fit the job and fails to start the job on time, and the job will miss the deadline.

Finally, `DistMax` completes fewer overall workload, as constructing the distribution using the maximal resource usage distorts the distribution estimates without providing any clear benefits.

5.4.3 Benefit attribution



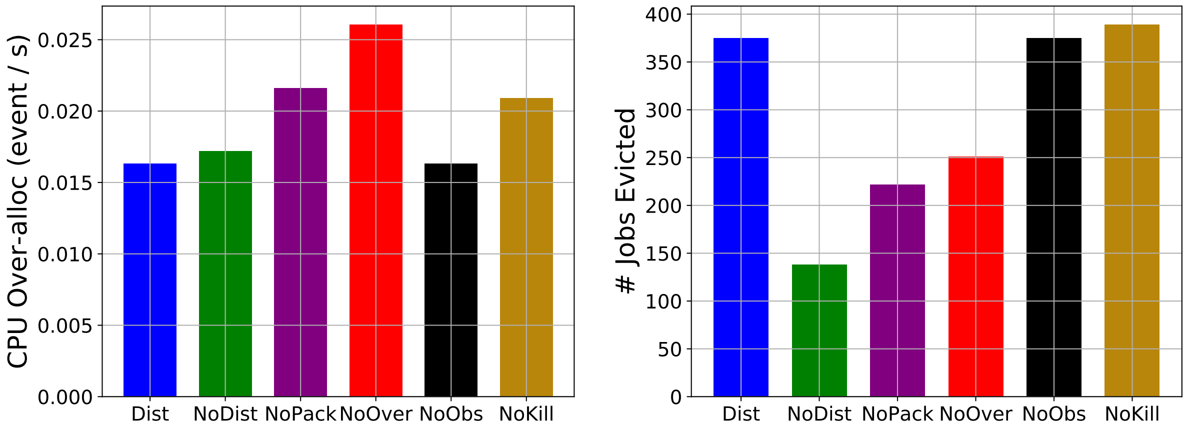
(a) SLO Miss Rate (b) SLO Goodput



(c) BE Goodput (d) Avg BE Latency

Figure 5.7: Compares the performance of `Dist` with the systems with individual features disabled in the simulated cluster.

This section evaluates the performance of `Dist` running in different configurations to evaluate the benefits from (a) distribution-based scheduling (`NoDist`), (b) optimistic over-packing



(a) Rate of Cpu over-allocation (b) Job evictions

Figure 5.8: Compares the amount of resource contention as `Dist` with the systems with individual features disabled.

(`NoPack`), (c) runtime over-estimate handling (`NoOver`), (d) using distribution from the observed resource usage (`NoObs`), and (e) preemption (`NoPreempt`) by having each of the features individually disabled. Figs. 5.7 and 5.8 compares the performance of these five configurations.

Disabling distribution-based scheduling results in 18% increase in the SLO miss-rate and 15% reduction in the SLO Goodput, even though `NoDist` has access to the other mitigation techniques available in the `Dist`. This clearly shows that having more detailed information of resource usage is beneficial to the scheduler in making informed scheduling decisions.

On the other hand, disabling optimistic over-packing showed mixed results. It showed slightly better SLO miss-rate compared to `Dist`, as the scheduler is more conservative in the placement – it will never schedule a task if there is any possibility of over-allocation – but 7% reduction in the SLO goodput. `Dist`'s optimistic over-packing allows the scheduler to get more work done, as it admitted larger jobs into the system, but at the same time increases a resource contention that may cause some smaller jobs to miss the deadline.

Runtime over-estimate handling is the part of the `Dist` that contributes most to its success. Disabling the feature `NoOver` dramatically increased the SLO miss-rate beyond the levels of `Point`, as the scheduler became too conservative in starting the jobs that are not predicted to finish on time. This clearly shows that just scheduling with the distribution is enough. There needss to be mechanisms that can deal with the potential mis-estimates in the prediction.

Disabling the feature that allowed scheduler to use the distribution from the observed resource usage did not result in any performance degradation, as I had expected. I suspect two possible reasons. First, the tasks that would have benefited using observed resource usage distribution may have been preempted or evicted before the cluster manager could collect enough execution history to trigger the update. Second, in the case of CPU usage distribution, the task could have been throttled already. Thus, the updated distribution may not be an accurate representation of the task's intrinsic CPU usage.

Looking at the rate of the CPU throttling and number of task being evicted, it is clear that the preemption feature works as intended, reducing the level of CPU contention in the cluster. However, it has not improved any of the core success metrics like the SLO miss-rate or goodput. I suspect the speed up from the relieving resource contention was not enough to make enough room in the cluster to run more jobs. Preemption also does not improve the memory contention much as tasks will be killed instantly upon a memory contention.

5.4.4 Sensitivity to the cluster size

Fig. 5.9 shows performance as a function of the cluster size as I inject the identical workload into the system. Note, the workload has an offered load of 0.8 on a 700-machine cluster. This means all datapoints smaller than 700 machines are overloaded clusters, and there is not enough resource to schedule all submitted jobs. As the cluster size increases beyond 700 machines, utilization decreases.

The SLO miss rate decreases for all systems as the cluster size grows, because the scheduling problem is easier when more resource is available to schedule jobs. Interestingly, the relative gap between `Point` and other systems widens as the cluster size increases. `Point` cannot fully make use of additional resource, because it is too often conservative and does not start as many jobs as other system as it still does not consider seemingly impossible jobs. This is confirmed by looking at the number of jobs that missed the deadline due to the scheduler not giving any chance for the jobs to run at all (Fig. 5.9(b)). On the other hand, `Dist` makes use of distributions to more robustly evaluate the likelihood of completing on time and the runtime over-estimate handling technique to try more jobs when idle resource is available. The gap continues to increase beyond the 700 machines datapoint, albeit marginally, as `Dist` and `Actual` can continue to use additional capacity more efficiently during a transient overload to meet deadlines.

Likewise, SLO goodput and BE goodput increase rapidly for all schedulers until the cluster size reaches 700 machines, because the schedulers can easily fill the newly available capacity with jobs that could not be scheduled before. The increase in goodput gradually levels off after 700 machines as the schedulers can only benefit from additional capacity in the case of transient overloads.

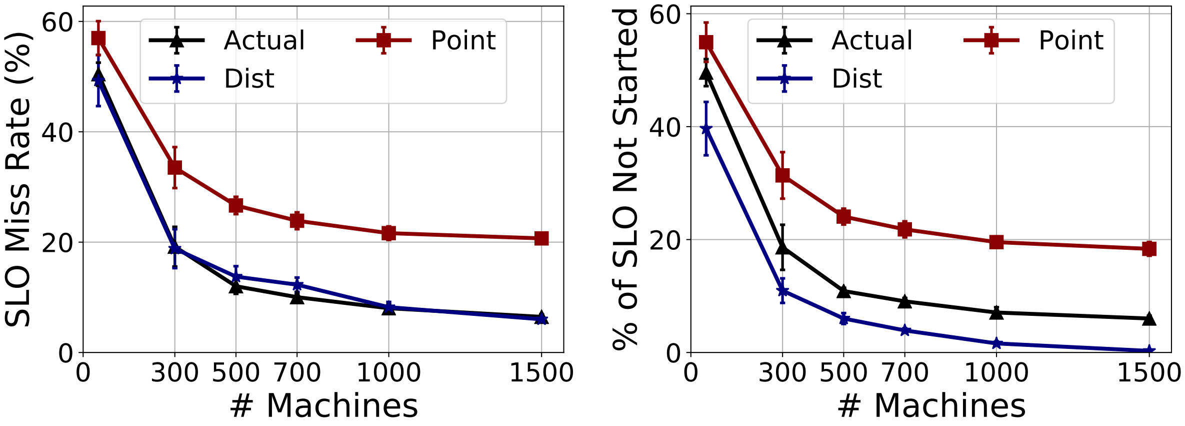
Latencies for BE jobs for all systems increase as the cluster size increases (except for an outlier at the 50 machines datapoint), as additional cluster capacity allows systems to run more BE jobs, especially those jobs that waited longer in the queue. BE latency spikes at the 50 machines datapoint, as it is the only cluster that does not have the capacity to even run just SLO jobs, and BE jobs are only scheduled when there is a transient underload.

5.4.5 Scheduler scalability

This section demonstrates that `Dist` can handle additional complexity from distribution-based scheduling with marginal computational overhead. Fig. 5.10 compares the distribution of scheduling cycle runtime for the four systems that use the average CPU usage as a basis for prediction. `Dist` requires more computation time to make decisions than the point prediction based systems, as it needs to consider possible outcomes of the distribution for the computation of (1) the aggregate resource usage of each node (Fig. 5.10(a)) and (2) the expected utility of each job (Fig. 5.10(b)). I observe that distribution-based scheduling only results in a moderate increase in the runtime for both operations. Although distribution-based scheduling requires the scheduler to consider more possible outcomes, a set of optimizations introduced in Sec. 5.2.3 has successfully brought down the computation overhead to a tractable level. I also note that overall runtime (Part 1 + Part 2) is much less than the frequency of the scheduling cycles (15s).

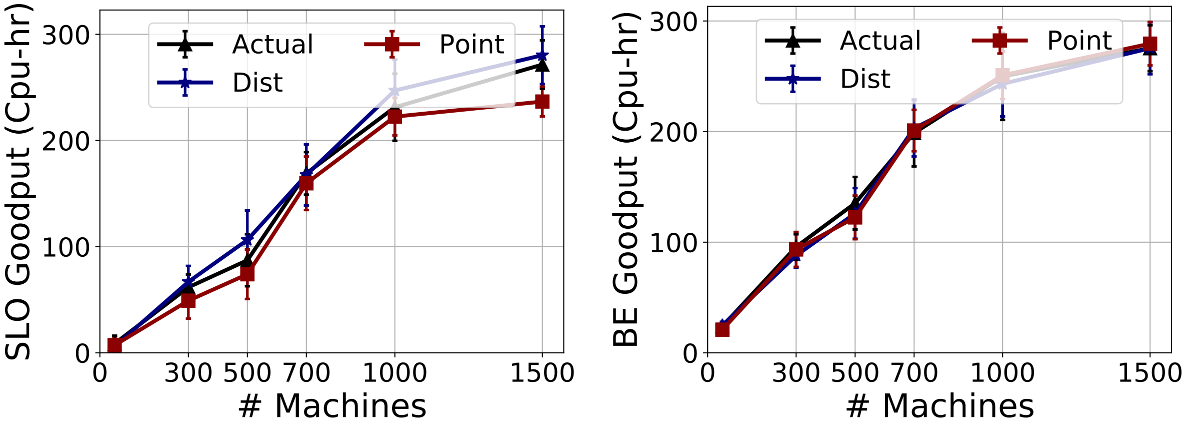
5.5 Summary

The resource-runtime distribution based scheduler DistSched uses resource usage distribution and mitigation mechanisms to exploit job history robustly. Experiments with 700 node simulation cluster demonstrate that the scheduler outperforms a state-of-the-art point-estimate scheduler, approaching the performance of a hypothetical scheduler operating with actual resource usage.



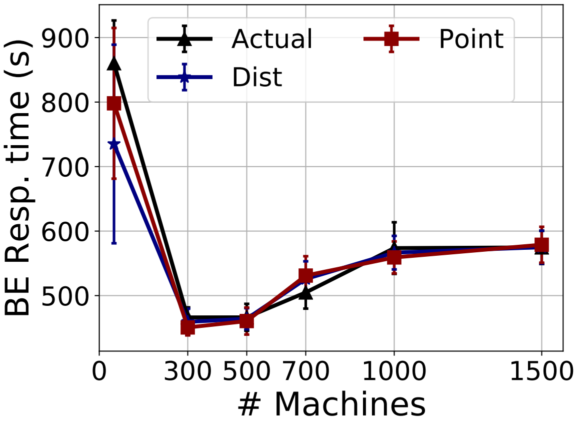
(a) SLO Miss Rate

(b) SLO Not Started



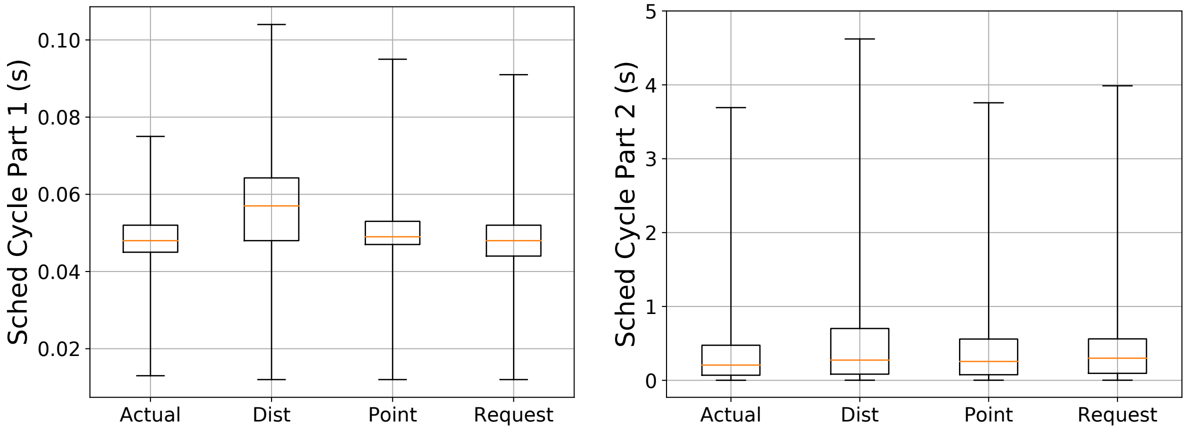
(c) SLO Goodput

(d) BE Goodput



(e) Avg BE Latency

Figure 5.9: Compares the performance of `Dist` with other systems in the simulated cluster. `Dist` consistently outperforms `Point` on SLO miss-rate and SLO Goodput while nearly matching `Point` and `Request` on BE Goodput and BE Latency.



(a) Scheduling Cycle Part 1

(b) Scheduling Cycle Part 2

Figure 5.10: Compares the amount of resource contention as `Dist` and other systems schedule jobs in the cluster. `Dist` outperforms `Point` on SLO miss-rate and SLO Goodput while nearly matching `Point` on BE Goodput and BE Latency.

Chapter 6

Conclusion

This dissertation demonstrates that schedulers that rely on information about job runtimes and resource usages can more robustly address imperfect predictions by looking at likelihoods of possible outcomes rather than single point expected outcomes and the allocation decision. To support this thesis statement, I did a workload analysis case study and designed two case study scheduling systems.

First, I did an analysis of four different traces from two different environments (a hedge fund and a national laboratory), contrasting their job characteristics, workload heterogeneity, resource utilization, and failure rates to the Google trace. The characterization suggests that there exists an inherent variability in the job runtimes and resource usage that is difficult to be captured by the point estimates. This result is highlighted through an evaluation of a history-based runtime predictor, JVuPredict, as it struggles to provide perfect runtime estimates, especially provided with insufficient information about the job.

Second, I designed 3Sigma, a scheduler that looks at runtime distributions instead of point-estimates to robustly cope with runtime mis-predictions. Extensive experiments on a real 256-node YARN cluster and in simulation demonstrate that 3Sigma's distribution-based scheduling greatly outperforms a state-of-the-art point estimate scheduler, approaching the performance of a hypothetical scheduler operating with perfect runtime estimates.

Third, I designed a resource-runtime distribution scheduler that looks at distributions of resource usage instead of user-provided point-estimates to robustly cope with resource usage uncertainty. Experiments in a 700 node simulation show that the scheduler, equipped with mis-estimation mitigation techniques, outperforms point-estimate based systems and approaches the performance of a hypothetical system with perfect resource usage estimates.

6.1 Future work

This section discusses potential future research directions for extending the thesis work.

6.1.1 Making better use of current resource observation

The resource-runtime based scheduler only uses the observation of resource usage of the running tasks in limited ways. Namely, it lacks a mechanism to apply the information to update the resource usage distribution estimate from the predictor, unlike runtime distribution where an observation (elapsed time) can be used to query the relevant subset of the distribution. An interesting research direction is to explore other ways to make use of the current resource usage to improve the current distribution of resource usage. For example, if a predicted distribution is multi-modal, an observation may be used to select a peak amongst the candidate peaks.

6.1.2 Exploiting patterns of resource usage

Currently, the resource usage distribution predictor only constructs a distribution based on a statistic (average or maximum) of the job resource usage for each job, instead of leveraging the entire time series of resource usage during the course of jobs' runtime. It would be interesting to study how much more information beyond a statistic can be feasibly captured and predicted by the predictor, as well as how can the scheduler leverage this information robustly.

6.1.3 Dependency-aware scheduling

Workflows, collection of jobs associated by input-output dependencies, are increasingly common in data-centers. For example, MapReduce jobs can be viewed as two-stage workflows and Spark applications can be viewed as multiple stage workflows. Systems such as [39, 40] designed for workflows exist, but they only rely on point-estimates and are expected to cope poorly with realistic estimate error profiles. It would be interesting to evaluate the distribution-based scheduling approach for scheduling workflows, especially if some workflows have a completion deadline.

6.1.4 Public Cloud or Hybrid Cloud environments

This dissertation assumes a traditional data-center setting, where the owners are operating a data-center in-house. More organizations, especially smaller enterprises, make use of the public cloud, where more computing resource can be acquired on-demand, or operate in a hybrid cloud setting, a mixture of both worlds. It would be interesting to evaluate the efficacy of the distribution-based scheduling approaches in Public Cloud or Hybrid Cloud settings.

6.1.5 Greedy scheduling algorithms

DistSched applies distribution-based scheduling ideas to one particular greedy algorithm: one that schedules one job at a time in the order of highest expected utility. However, in the past, the cluster scheduling research community has explored a wide variety of metrics and heuristics to schedule jobs in a greedy manner, namely EDF, SJF, etc. It would be interesting to explore and evaluate potential ways to choose the order in which pending jobs are considered for scheduling in the context of distribution-based scheduling systems.

6.1.6 Utility functions

This dissertation assumes a particular type of workload, namely a mixture of service level objective (SLO) jobs with deadlines and latency sensitive best-effort jobs, and their corresponding utility functions. However, it does not explore other interesting aspects related to the utility function. One area for continuing research is exploring different forms of utility functions that may be applicable to jobs with different requirements and whether there exists a general way of coming up with a utility function. Another promising research area is to explore ways to incentivize the users to provide their true utility function. If the system naively allows users to submit their utility functions, it is entirely possible for the users to deliberately provide false information to their advantage (e.g., schedule their jobs before everyone else). For example, associating a monetary value (dollar cost) may be a solution, but it may be difficult for users to evaluate the potential monetary benefits of running a particular job.

6.1.7 Adapting to drift and trend in the history

3σPredict and `DSPredict` assume a newly arriving job will be similar in terms of resource usage and runtime to at least some subset of jobs run in the past. However, in reality, even explicitly recurring jobs can evolve over time. For example, more users may use a system over time and increase the runtime of a job analyzing the log data coming from the system. 3σPredict and `DSPredict` use recency-based experts that use only a few recent datapoints. This addresses the issue to some degree but is not a perfect solution. For example, it may take an arbitrarily long time for an expert to converge if the job is infrequently executed or there is a drastic change in the job behavior (e.g., major version update). It is interesting to explore other mechanisms, such as explicitly clearing a part of the history, that a predictor may use to adapt to rapid shifts in jobs' behavior.

Bibliography

[1] What are FPGAs and Project Brainwave. `https://docs.microsoft.com/en-us/azure/machine-learning/service/concept-accelerate-with-fpgas`. 2.1

[2] Amazon EC2 F1 Instances. `https://aws.amazon.com/ec2/instance-types/f1/`. 2.1

[3] OpenStack. `https://www.openstack.org/`. 2.1

[4] Azure Virtual Machines. `https://azure.microsoft.com/en-us/services/virtual-machines/`, 2018. 2.1

[5] AWS EC2. `http://aws.amazon.com/ec2/`, 2018. 2.1

[6] Google Compute Engine. `https://cloud.google.com/compute/`, 2018. 2.1

[7] Adaptive Computing. MOAB HPC Suite. `http://www.adaptivecomputing.com/products/hpc-products/moab-hpc-basic-edition/`. 2.4.4

[8] George Amvrosiadis, Jun Woo Park, Gregory R Ganger, Garth A Gibson, Elisabeth Baseman, and Nathan DeBardeleben. On the diversity of cluster workloads and its impact on research results. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, pages 533–546, 2018. 1, 1.1, 1.2, 1.3, 5, 5.1.2

[9] Ganesh Ananthanarayanan, Ali Ghodsi, Scott Shenker, and Ion Stoica. Effective Straggler Mitigation: Attack of the Clones. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 185–198, Lombard, IL, 2013. USENIX. ISBN 978-1-931971-00-3. URL `https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/ananthanarayanan`. 3.2, 3.2

[10] Salman A Baset, Long Wang, and Chunqiang Tang. Towards an understanding of oversubscription in cloud. In *Presented as part of the 2nd {USENIX} Workshop on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services*, 2012. 5, 5.1.2

[11] Yael Ben-Haim and Elad Tom-Tov. A streaming parallel decision tree algorithm. *Journal of Machine Learning Research*, 11(Feb):849–872, 2010. 4.3.1, 5.3.1

[12] Raouf Boutaba, Lu Cheng, and Qi Zhang. On cloud computational models and the heterogeneity challenge. *Journal of Internet Services and Applications*, 3(1):77–86, 2012. 2.1

[13] Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou, Zhengping Qian, Ming

Wu, and Lidong Zhou. Apollo: Scalable and coordinated scheduling for cloud-scale computing. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 285–300, Broomfield, CO, October 2014. USENIX Association. ISBN 978-1-931971-16-4. URL `https://www.usenix.org/conference/osdi14/technical-sessions/presentation/boutin`. 1, 2.2, 2.3, 3.6.1, 4, 4.1

[14] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. Borg, omega, and kubernetes. 2016. 2, 2.1, 2.3, 5.1, 5.1.2, 5.1.3, 5.4.1

[15] Anton Burtsev, Prashanth Radhakrishnan, Mike Hibler, and Jay Lepreau. Transparent checkpoints of closed distributed systems in emulab. In *Proceedings of the 4th ACM European Conference on Computer Systems*, EuroSys '09, pages 173–186, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-482-9. doi: 10.1145/1519065.1519084. URL `http://doi.acm.org/10.1145/1519065.1519084`. 3.1

[16] Ignacio Cano, Srinivas Aiyar, and Arvind Krishnamurthy. Characterizing Private Clouds: A Large-Scale Empirical Analysis of Enterprise Clusters. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, SoCC '16, pages 29–41, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4525-5. doi: 10.1145/2987550.2987584. URL `http://doi.acm.org/10.1145/2987550.2987584`. 3.8

[17] Marcus Carvalho, Walfredo Cirne, Franciso Brasileiro, and John Wilkes. Long-term SLOs for reclaimed cloud computing resources. In *ACM Symposium on Cloud Computing (SoCC)*, pages 20:1–20:13, Seattle, WA, USA, 2014. URL `http://dl.acm.org/citation.cfm?id=2670999`. 2.1, 3.4

[18] Chen Chen, Wei Wang, Shengkai Zhang, and Bo Li. Cluster Fair Queueing: Speeding up Data-Parallel Jobs with Delay Guarantees. In *Proceedings of the IEEE International Conference on Computer Communications*, IEEE INFOCOM 2017, May 2017. 3.3

[19] S. Chen, M. Ghorbani, Y. Wang, P. Bogdan, and M. Pedram. Trace-Based Analysis and Prediction of Cloud Computing User Behavior Using the Fractal Modeling Technique. In *2014 IEEE International Congress on Big Data*, pages 733–739, June 2014. doi: 10.1109/BigData.Congress.2014.108. 3.2

[20] X. Chen, C. D. Lu, and K. Pattabiraman. Failure Analysis of Jobs in Compute Clouds: A Google Cluster Case Study. In *2014 IEEE 25th International Symposium on Software Reliability Engineering*, pages 167–177, Nov 2014. doi: 10.1109/ISSRE.2014.34. 3.5, 3.5

[21] Yanpei Chen, Sara Alspaugh, and Randy Katz. Interactive Analytical Processing in Big Data Systems: A Cross-Industry Study of MapReduce Workloads. In *Proceedings of the VLDB Endowment*, PVLDB, 2012. 2.4, 3, 3.4, 3.4, 3.8

[22] Yun Chi, Hakan Hacígümüş, Wang-Pin Hsiung, and Jeffrey F Naughton. Distribution-based query scheduling. *Proceedings of the VLDB Endowment*, 6(9):673–684, 2013. 4.1.2

[23] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *ACM SIGOPS operating systems review*. ACM, 2017. 2.3, 3.8

[24] Carlo Curino, Djellel E. Difallah, Chris Douglas, Subru Krishnan, Raghu Ramakrishnan, and Sriram Rao. Reservation-based scheduling: If you're late don't blame us! In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, pages 2:1–2:14, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3252-1. doi: 10.1145/2670979.2670981. URL `http://doi.acm.org/10.1145/2670979.2670981`. 1, 2.2, 2.3, 3.6.1, 4, 4.1, 4.4.1, 4.4.6, 5.1

[25] Pamela Delgado, Florin Dinu, Anne-Marie Kermarrec, and Willy Zwaenepoel. Hawk: Hybrid datacenter scheduling. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '15, pages 499–510, Berkeley, CA, USA, 2015. USENIX Association. ISBN 978-1-931971-225. URL `http://dl.acm.org/citation.cfm?id=2813767.2813804`. 2.3, 3.2, 3.2

[26] Pamela Delgado, Diego Didona, Florin Dinu, and Willy Zwaenepoel. Job-aware Scheduling in Eagle: Divide and Stick to Your Probes. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, SoCC '16, pages 497–509, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4525-5. doi: 10.1145/2987550.2987563. URL `http://doi.acm.org/10.1145/2987550.2987563`. 3.2, 3.2

[27] Pamela Delgado, Diego Didona, Florin Dinu, and Willy Zwaenepoel. Kairos: Preemptive data center scheduling without runtime estimates. In *Proceedings of the 9th ACM Symposium on Cloud Computing*, number CONF, 2018. 2.3

[28] Christina Delimitrou and Christos Kozyrakis. Quasar: Resource-efficient and qos-aware cluster management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 127–144, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2305-5. doi: 10.1145/2541940.2541941. URL `http://doi.acm.org/10.1145/2541940.2541941`. 2.1, 2.2, 3.4, 3.4, 3.4

[29] Eric Eide, Leigh Stoller, and Jay Lepreau. An Experimentation Workbench for Replayable Networking Research. In *Proceedings of the 4th USENIX Conference on Networked Systems Design &#38; Implementation*, NSDI'07, pages 16–16, Berkeley, CA, USA, 2007. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=1973430.1973446`. 3.1

[30] N. El-Sayed, H. Zhu, and B. Schroeder. Learning from Failure Across Multiple Clusters: A Trace-Driven Approach to Understanding, Predicting, and Mitigating Job Terminations. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 1333–1344, June 2017. doi: 10.1109/ICDCS.2017.317. 3.5, 3.5, 3.5, 3.5

[31] Dror G Feitelson, Larry Rudolph, and Uwe Schwiegelshohn. Parallel job scheduling—a status report. In *Job Scheduling Strategies for Parallel Processing*, pages 1–16. Springer, 2004. 3.6.1

[32] Dror G. Feitelson, Dan Tsafrir, and David Krakov. Experience with using the Parallel Workloads Archive. *Journal of Parallel and Distributed Computing*, 74(10):2967 – 2982, 2014. doi: http://dx.doi.org/10.1016/j.jpdc.2014.06.013. URL `http://www.sciencedirect.com/science/article/pii/S0743731514001154`. 2.4,

3, 3.8

[33] Andrew D. Ferguson, Peter Bodik, Srikanth Kandula, Eric Boutin, and Rodrigo Fonseca. Jockey: guaranteed job latency in data parallel clusters. In *Proc. of the 7th ACM european conference on Computer Systems*, EuroSys '12, pages 99–112, 2012. ISBN 978-1-4503-1223-3. doi: 10.1145/2168836.2168847. URL `http://doi.acm.org/10.1145/2168836.2168847`. 2.2, 2.3, 3.6.1, 5, 5.1

[34] P. Garraghan, I. S. Moreno, P. Townend, and J. Xu. An Analysis of Failure-Related Energy Waste in a Large-Scale Cloud Environment. *IEEE Transactions on Emerging Topics in Computing*, 2(2):166–180, June 2014. ISSN 2168-6750. doi: 10.1109/TETC.2014.2304500. 3.5, 3.5

[35] Bogdan Ghit and Dick Epema. Better Safe Than Sorry: Grappling with Failures of In-Memory Data Analytics Frameworks. In *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '17, pages 105–116, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4699-3. doi: 10.1145/3078597.3078600. URL `http://doi.acm.org/10.1145/3078597.3078600`. 3.5, 3.5

[36] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant resource fairness: fair allocation of multiple resource types. In *Proc. of the 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI'11)*, 2011. 5.1

[37] Ionel Gog, Malte Schwarzkopf, Adam Gleave, Robert N. M. Watson, and Steven Hand. Firmament: Fast, Centralized Cluster Scheduling at Scale. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 99–115, GA, 2016. USENIX Association. ISBN 978-1-931971-33-1. 3.3, 3.3

[38] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. Multi-resource packing for cluster schedulers. *ACM SIGCOMM Computer Communication Review*, 44(4):455–466, 2015. 4.1, 5.1

[39] Robert Grandl, Mosharaf Chowdhury, Aditya Akella, and Ganesh Ananthanarayanan. Altruistic scheduling in multi-resource clusters. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 65–80, 2016. 6.1.3

[40] Robert Grandl, Srikanth Kandula, Sriram Rao, Aditya Akella, and Janardhan Kulkarni. {GRAPHENE}: Packing and dependency-aware scheduling for data-parallel clusters. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 81–97, 2016. 1, 3.6.1, 4.1, 4.1.1, 5.1, 6.1.3

[41] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. Efficient Memory Disaggregation with Infiniswap. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 649–667, Boston, MA, 2017. USENIX Association. ISBN 978-1-931971-37-9. URL `https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/gu`. 3.4

[42] Mor Harchol-Balter. *Network analysis without explonentiality assumptions*. University of California, Berkeley, 1996. 5.3.2

[43] Mor Harchol-Balter and Allen B. Downey. Exploiting process lifetime distributions for dynamic load balancing. *ACM Trans. Comput. Syst.*, 15(3):253–285, August 1997. ISSN 0734-2071. doi: 10.1145/263326.263344. URL `http://doi.acm.org/10.1145/263326.263344`. 2.3

[44] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2011. ISBN 012383872X, 9780123838728. 2.4.1

[45] Mike Hibler, Robert Ricci, Leigh Stoller, Jonathon Duerig, Shashi Guruprasad, Tim Stack, Kirk Webb, and Jay Lepreau. Large-scale virtualization in the emulab network testbed. In *USENIX 2008 Annual Technical Conference*, ATC'08, pages 113–128, Berkeley, CA, USA, 2008. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=1404014.1404023`. 3.1

[46] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A.D. Joseph, R.H. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *Proc. of the 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI'11)*, 2011. 2.1, 2.3, 2.4.2

[47] Chien-Chun Hung, Leana Golubchik, and Minlan Yu. Scheduling Jobs Across Geo-distributed Datacenters. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, SoCC '15, pages 111–124, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3651-2. doi: 10.1145/2806777.2806780. URL `http://doi.acm.org/10.1145/2806777.2806780`. 3.4, 3.4

[48] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 59–72, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-636-3. doi: 10.1145/1272996.1273005. URL `http://doi.acm.org/10.1145/1272996.1273005`. 3.6.1

[49] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. Quincy: Fair scheduling for distributed computing clusters. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 261–276, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-752-3. doi: 10.1145/1629575.1629601. URL `http://doi.acm.org/10.1145/1629575.1629601`. 3.3, 3.3, 5, 5.1

[50] Mohammad Islam, Angelo K Huang, Mohamed Battisha, Michelle Chiang, Santhosh Srinivasan, Craig Peters, Andreas Neumann, and Alejandro Abdelnur. Oozie: Towards a Scalable Workflow Management System for Hadoop. In *SWEET Workshop*, 2012. 2.2, 4.1

[51] Sadeka Islam, Jacky Keung, Kevin Lee, and Anna Liu. Empirical prediction models for adaptive resource provisioning in the cloud. *Future Generation Computer Systems*, 28(1): 155–162, 2012. 2.3

[52] Virajith Jalaparti, Peter Bodik, Ishai Menache, Sriram Rao, Konstantin Makarychev, and Matthew Caesar. Network-aware scheduling for data-parallel jobs: Plan when you can.

In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 407–420, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3542-3. doi: 10.1145/2785956.2787488. URL `http://doi.acm.org/10.1145/2785956.2787488`. 2.3

[53] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–12. IEEE, 2017. 2.1

[54] Sangeetha Abdu Jyothi, Carlo Curino, Ishai Menache, Shravan Matthur Narayanamurthy, Alexey Tumanov, Jonathan Yaniv, Ruslan Mavlyutov, Íñigo Goiri, Subru Krishnan, Janardhan Kulkarni, et al. Morpheus: towards automated slos for enterprise clusters. In *Proceedings of the 12th USENIX conference on Operating Systems Design and Implementation*, pages 117–134. USENIX Association, 2016. 1, 2.2, 2.3, 3.6.1, 4, 4.1, 4.1.2, 4.4.1, 5, 5.1, 5.1.2

[55] S. Krishnaswamy, S. W. Loke, and A. Zaslavsky. Estimating computation times of data-intensive applications. *IEEE Distributed Systems Online*, 5(4), April 2004. ISSN 1541-4922. doi: 10.1109/MDSO.2004.1301253.

[56] Shonali Krishnaswamy, Seng Wai Loke, and Arkady Zaslavsky. Estimating computation times of data-intensive applications. In *IEEE Distributed Systems Online*. IEEE, 2010. 2.3

[57] Jacob Leverich and Christos Kozyrakis. Reconciling High Server Utilization and Sub-millisecond Quality-of-service. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 4:1–4:14, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2704-6. doi: 10.1145/2592798.2592821. URL `http://doi.acm.org/10.1145/2592798.2592821`. 2.1, 2.2, 3.4, 3.4

[58] J. Liu, H. Shen, and L. Chen. CORP: Cooperative Opportunistic Resource Provisioning for Short-Lived Jobs in Cloud Systems. In *2016 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 90–99, Sept 2016. doi: 10.1109/CLUSTER.2016.65. 2.1, 2.2, 3.4

[59] Shuo Liu, Gang Quan, and Shangping Ren. On-line scheduling of real-time services for cloud computing. In *Services (SERVICES-1), 2010 6th World Congress on*. IEEE, 2010. 4.1.2

[60] Jiuyue Ma, Xiufeng Sui, Ninghui Sun, Yupeng Li, Zihao Yu, Bowen Huang, Tianni Xu, Zhicheng Yao, Yun Chen, Haibin Wang, Lixin Zhang, and Yungang Bao. Supporting Differentiated Services in Computers via Programmable Architecture for Resourcing-on-Demand (PARD). In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 131–143, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-2835-7. doi: 10.1145/2694344.2694382. URL `http://doi.acm.org/10.1145/2694344.2694382`. 2.1, 3.4

[61] Paul Marshall, Kate Keahey, and Tim Freeman. Improving Utilization of Infrastructure Clouds. In *Proceedings of the 2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, CCGRID '11, pages 205–214, Washington, DC, USA, 2011.

IEEE Computer Society. ISBN 978-0-7695-4395-6. doi: 10.1109/CCGrid.2011.56. URL `http://dx.doi.org/10.1109/CCGrid.2011.56`. 2.1, 3.4

[62] Frank J. Massey Jr. The Kolmogorov-Smirnov Test for Goodness of Fit. *Journal of the American Statistical Association*, 46(253):68–78, 1951. 3.4

[63] Esteban Meneses, Xiang Ni, Terry Jones, and Don Maxwell. Analyzing the Interplay of Failures and Workload on a Leadership-Class Supercomputer. In *Cray User Group Conference (CUG)*, April 2015. 3.3

[64] Kristi Morton, Abram Friesen, Magdalena Balazinska, and Dan Grossman. Estimating the progress of mapreduce pipelines. In *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*, pages 681–684. IEEE, 2010. 2.3

[65] I. A. Moschakis and H. D. Karatza. Performance and cost evaluation of gang scheduling in a cloud computing system with job migrations and starvation handling. In *Computers and Communications (ISCC), 2011 IEEE Symposium on*, pages 418–423, June 2011. doi: 10.1109/ISCC.2011.5983873. 2.2, 4.1

[66] Office of Science (DOE-SC) and the National Nuclear Security Administration (NNSA), U.S. Department of Energy. Exascale Computing Project. `https://exascaleproject.org/`. 3.3

[67] Jennifer Ortiz, Brendan Lee, Magdalena Balazinska, Johannes Gehrke, and Joseph L Hellerstein. SLAOrchestrator: reducing the cost of performance SLAs for cloud data analytics. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, pages 547–560, 2018. 2.3

[68] John K Ousterhout. Scheduling techniques for concurrent systems. In *International Conference on Distributed Computing Systems (ICDCS)*, volume 82, pages 22–30, 1982. 2.2, 4.1

[69] Jun Woo Park, Alexey Tumanov, Angela Jiang, Michael A. Kozuch, and Gregory R. Ganger. 3sigma: Distribution-based cluster scheduling for runtime uncertainty. In *Proceedings of the Thirteenth European Conference on Computer Systems*, EuroSys '18, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5584-1/18/04. doi: 10.1145/3190508.3190515. URL `http://doi.acm.org/10.1145/3190508.3190515`. 1.1, 1.2, 1.3

[70] Kaushik Rajan, Dharmesh Kakadia, Carlo Curino, and Subru Krishnan. Perforator: Eloquent performance models for resource optimization. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, SoCC '16, pages 415–427, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4525-5. doi: 10.1145/2987550.2987566. URL `http://doi.acm.org/10.1145/2987550.2987566`. 2.3

[71] S. Rampersaud and D. Grosu. Sharing-Aware Online Virtual Machine Packing in Heterogeneous Resource Clouds. *IEEE Transactions on Parallel and Distributed Systems*, 28(7): 2046–2059, July 2017. ISSN 1045-9219. doi: 10.1109/TPDS.2016.2641937. 3.2

[72] Charles Reiss, Alexey Tumanov, Gregory R Ganger, Randy H Katz, and Michael A Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proceedings of the Third ACM Symposium on Cloud Computing*, page 7. ACM, 2012. (document), 1, 2.1,

2.4, 2.4.1, 2.1, 3, 3.4, 3.4, 4.1, 4.1, 4.4.1, 5, 5.4.1

[73] Kai Ren, YongChul Kwon, Magdalena Balazinska, and Bill Howe. Hadoop's Adolescence: An Analysis of Hadoop Usage in Scientific Workloads. *Proc. VLDB Endow.*, 6(10):853–864, August 2013. ISSN 2150-8097. doi: 10.14778/2536206.2536213. URL `http://dx.doi.org/10.14778/2536206.2536213`. 3.8

[74] A. Rosà, L. Y. Chen, and W. Binder. Predicting and Mitigating Jobs Failures in Big Data Clusters. In *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 221–230, May 2015. doi: 10.1109/CCGrid.2015.139. 3.5

[75] A. Rosà, L. Y. Chen, and W. Binder. Understanding the Dark Side of Big Data Clusters: An Analysis beyond Failures. In *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 207–218, June 2015. doi: 10.1109/DSN.2015.37. 3.5

[76] SchedMD. SLURM Workload Manager. `https://slurm.schedmd.com/`. 2.4.3

[77] Jennifer M. Schopf and Francine Berman. Stochastic scheduling. In *SC '99 Proceedings of the 1999 ACM/IEEE conference on Supercomputing*. ACM, 1999. 4.1.2

[78] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. Omega: flexible, scalable schedulers for large compute clusters. In *ACM Eurosys Conference*, 2013. 2.4.1, 3.3, 3.3

[79] Bikash Sharma, Victor Chudnovsky, Joseph L. Hellerstein, Rasekh Rifaat, and Chita R. Das. Modeling and synthesizing task placement constraints in Google compute clusters. In *Proc. of the 2nd ACM Symposium on Cloud Computing*, SOCC '11, pages 3:1–3:14. ACM, 2011. ISBN 978-1-4503-0976-9. doi: http://doi.acm.org/10.1145/2038916.2038919. URL `http://doi.acm.org/10.1145/2038916.2038919`. 4.1

[80] S. Shen, V. v. Beek, and A. Iosup. Statistical Characterization of Business-Critical Workloads Hosted in Cloud Datacenters. In *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 465–474, May 2015. doi: 10.1109/CCGrid.2015.60. 3.8

[81] Warren Smith, Ian T. Foster, and Valerie E. Taylor. Predicting application run times using historical information. In *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*. IEEE, 1998. 2.3, 4.1.1

[82] Marc Snir, Robert W Wisniewski, Jacob A Abraham, Sarita V Adve, Saurabh Bagchi, Pavan Balaji, Jim Belak, Pradip Bose, Franck Cappello, Bill Carlson, Andrew A Chien, Paul Coteus, Nathan A DeBardeleben, Pedro C Diniz, Christian Engelmann, Mattan Erez, Saverio Fazzari, Al Geist, Rinku Gupta, Fred Johnson, Sriram Krishnamoorthy, Sven Leyffer, Dean Liberty, Subhasish Mitra, Todd Munson, Rob Schreiber, Jon Stearley, and Eric Van Hensbergen. Addressing failures in exascale computing. *The International Journal of High Performance Computing Applications*, 28(2):129–173, 2014. doi: 10.1177/1094342014522573. 3.5

[83] Roshan Sumbaly, Jay Kreps, and Sam Shah. The Big Data Ecosystem at LinkedIn. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*,

SIGMOD, 2013. 2.2, 4.1

[84] The Apache Software Foundation. Apache Spark. `https://spark.apache.org/`. 2.4.2

[85] P. Thinakaran, J. R. Gunasekaran, B. Sharma, M. T. Kandemir, and C. R. Das. Phoenix: A Constraint-Aware Scheduler for Heterogeneous Datacenters. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 977–987, June 2017. doi: 10.1109/ICDCS.2017.262. 3.2, 3.2

[86] Dan Tsafrir, Yoav Etsion, and Dror G. Feitelson. Backfilling using system-generated predictions rather than user runtime estimates. In *IEEE Transactions on Parallel and Distributed Systems*. IEEE, 2007. 4.1.2, 4.3.2

[87] Alexey Tumanov, Angela Jiang, Jun Woo Park, Michael A. Kozuch, and Gregory R. Ganger. JamaisVu: Robust Scheduling with Auto-Estimated Job Runtimes. Technical Report CMU-PDL-16-104, Carnegie Mellon University, September 2016. 1, 2.3, 3, 3.6, 3.6.1, 3.6.2, 4, 4.1.1

[88] Alexey Tumanov, Timothy Zhu, Jun Woo Park, Michael A. Kozuch, Mor Harchol-Balter, and Gregory R. Ganger. Tetrisched: Global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, pages 35:1–35:16, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4240-7. doi: 10.1145/2901318.2901355. URL `http://doi.acm.org/10.1145/2901318.2901355`. 1, 2.1, 2.2, 3.6.1, 4, 4, 4.1, 4.1.1, 4.3.3, 4.3.3, 4.4.1, 4.4.6, 5.1, 5.2.2, 5.2.2

[89] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, , Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. Apache Hadoop YARN: Yet another resource negotiator. In *Proc. of the 4th ACM Symposium on Cloud Computing*, SOCC '13, 2013. 2, 2.1, 2.3, 4.4.1, 5.1.3

[90] Shivaram Venkataraman, Zongheng Yang, Michael Franklin, Benjamin Recht, and Ion Stoica. Ernest: efficient performance prediction for large-scale advanced analytics. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 363–378, 2016. 2.3

[91] S. Verboven, P. Hellinckx, F. Arickx, and J. Broeckhove. Runtime prediction based grid scheduling of parameter sweep jobs. In *Asia-Pacific Services Computing Conference, 2008. APSCC '08. IEEE*, pages 33–38, Dec 2008. doi: 10.1109/APSCC.2008.189. 2.3

[92] A. Verma, M. Korupolu, and J. Wilkes. Evaluating job packing in warehouse-scale computing. In *2014 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 48–56, Sept 2014. doi: 10.1109/CLUSTER.2014.6968735. 1, 2.1, 3.6.1, 4, 4.1

[93] Abhishek Verma, Ludmila Cherkasova, and Roy H. Campbell. ARIA: Automatic resource inference and allocation for mapreduce environments. In *Proceedings of the 8th ACM International Conference on Autonomic Computing*, ICAC '11, pages 235–244, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0607-2. doi: 10.1145/1998582.1998637. URL `http://doi.acm.org/10.1145/1998582.1998637`.

[94] Abhishek Verma, Ludmila Cherkasova, and Roy H. Campbell. Aria: Automatic resource inference and allocation for mapreduce environments. In *Proceedings of the 8th ACM International Conference on Autonomic Computing*, ICAC '11, pages 235–244, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0607-2. doi: 10.1145/1998582.1998637. URL `http://doi.acm.org/10.1145/1998582.1998637`. 2.3, 3.6.1

[95] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, pages 18:1–18:17, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3238-5. doi: 10.1145/2741948.2741964. URL `http://doi.acm.org/10.1145/2741948.2741964`. 2.4.1, 4.1.2, 4.4.1

[96] W. Wang, B. Li, B. Liang, and J. Li. Multi-resource Fair Sharing for Datacenter Jobs with Placement Constraints. In *SC16: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1003–1014, Nov 2016. doi: 10.1109/SC.2016.85. 3.3, 3.3, 3.4

[97] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An Integrated Experimental Environment for Distributed Systems and Networks. In *Proc. of the Fifth Symposium on Operating Systems Design and Implementation*, pages 255–270, Boston, MA, December 2002. USENIX Association. 3.1

[98] John Wilkes. More Google cluster data. Google research blog, November 2011. Posted at `http://googleresearch.blogspot.com/2011/11/more-google-cluster-data.html`. 2.4.1, 3.5

[99] Timothy Wood, Prashant J Shenoy, Arun Venkataramani, and Mazin S Yousif. Black-box and gray-box strategies for virtual machine migration. In *NSDI*, volume 7, pages 17–17, 2007. 4.1.2

[100] Yulai Yuan, Yongwei Wu, Qiuping Wang, Guangwen Yang, and Weimin Zheng. Job failures in high performance computing systems: A large-scale empirical study. *Computers & Mathematics with Applications*, 63(2):365 – 377, 2012. ISSN 0898-1221. doi: http://dx.doi.org/10.1016/j.camwa.2011.07.040. 3.5

[101] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the 5th European conference on Computer systems (Eurosys)*, pages 265–278. ACM, 2010. 2.2, 4.1, 5, 5.1