

IMPROVING RESOURCE EFFICIENCY IN CLOUD COMPUTING

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF ELECTRICAL
ENGINEERING
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Christina Delimitrou

August 2015

© 2015 by Christina Delimitrou. All Rights Reserved.
Re-distributed by Stanford University under license with the author.



This work is licensed under a Creative Commons Attribution-Noncommercial 3.0 United States License.

<http://creativecommons.org/licenses/by-nc/3.0/us/>

This dissertation is online at: <http://purl.stanford.edu/mz539pv2699>

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Christos Kozyrakis, Primary Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

John Ousterhout

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Mendel Rosenblum

Approved for the Stanford University Committee on Graduate Studies.

Patricia J. Gumport, Vice Provost for Graduate Education

This signature page was generated electronically upon submission of this dissertation in electronic format. An original signed hard copy of the signature page is on file in University Archives.

Abstract

Cloud computing is at a critical juncture. An increasing amount of computation is now hosted in private and public clouds. At the same time, datacenter resource efficiency, i.e., the effective utility we extract from system resources has remained notoriously low, with utilization rarely exceeding 20-30%. Low utilization coupled with the lack of scaling in hardware due to technology limitations poses threatening scalability roadblocks for cloud computing.

At a high level, two main reasons hinder efficient scalability in datacenters. First, the reservation-based interface through which resources are currently allocated is fundamentally flawed. Users must determine how many resources a new application requires to meet its quality of service (QoS) constraints. Unfortunately this is extremely difficult for users that tend to overprovision their reservations, resulting in mostly-allocated, but lightly-utilized systems. Second, underutilization is aggravated by performance unpredictability; the result of heterogeneity in hardware platforms, interference between applications contending in shared resources, and spikes in input load. Unpredictability results in further resource overprovisioning by users.

The focus of this dissertation is to enable efficient, scalable and performance-aware datacenters with tens to hundreds of thousands of machines by improving cluster management. To this end, we present contributions that address both the system-user interface, and the complexity of resource management at scale. These techniques are directly applicable to current systems, with modest design alterations.

We first present a new declarative interface between users and cluster manager that centers around performance, instead of resource reservations. This enables users to focus on the high level performance objectives an application must meet, as opposed

to the intrinsics on how these objectives should be achieved using low level resources.

On the system side, we make two fundamental contributions. First, we design a practical system that leverages data mining to quickly understand the resource requirements of incoming applications in an online manner. We establish that resource management at this scale cannot be solved with the traditional trial-and-error approach of conventional architecture and system design. We show that instead we can introduce data mining principles which leverage the knowledge the system accumulates over time from incoming applications, to significantly benefit both performance and efficiency. We first use this approach in Paragon to tackle the platform heterogeneity and workload interference challenges in datacenter management. The cluster manager relies on collaborative filtering to identify the most suitable hardware platform for a new, unknown application and its sensitivity to interference in various shared resources. We then extend a similar approach to address the larger problem of resource assignment and resource allocation with Quasar. To ensure minimal management overheads, we decompose the problem to four dimensions; platform heterogeneity, application interference, resource scale-up and scale-out. This enables the majority of applications to meet their QoS targets, while operating at 70% utilization, on a cluster with several hundred servers. In contrast, a reservation-based system rarely exceeds 15-20% utilization, with worse per-application performance.

Our second contribution pertains to designing scalable scheduling techniques that use the information from Paragon and Quasar to perform efficient and QoS-aware resource allocations. We develop Tarcil, a scalable scheduler that reconciles the high quality of sophisticated centralized schedulers with the low latency of distributed sampling-based systems. Tarcil relies on a simple analytical framework to sample resources in a way that provides statistical guarantees on a job meeting its QoS constraints. It incurs a few milliseconds of scheduling overhead, making it appropriate for highly-loaded clusters, servicing both short- and long-running applications. Finally, we design HCloud, a resource provisioning system for public cloud providers. HCloud leverages the information on the resource preferences of applications to determine the type (e.g., reserved versus on-demand) and size of required instances. The system guarantees high application performance, while securing significant cost savings.

Acknowledgements

Many people are responsible for supporting me in completing this dissertation. First and foremost, I want to thank my advisor, Christos Kozyrakis, for his advice not only on research, but career paths and life in general. Working on a topic you are passionate about is one of the most important aspects of graduate studies, and Christos' support on selecting a project I found compelling has been invaluable; even if it had a lot of math. On a more personal note, I also want to thank him for being on several occasions family away from family.

I am also thankful to all the faculty members of the Stanford Experimental Data-center Lab (SEDCL) and the Pervasive Parallelism Lab (PPL), and especially Mendel Rosenblum and John Ousterhout for their advice, feedback and inspiration over the years, and for taking the time to be on my reading committee. I am also grateful to Bill Dally and Kunle Olukotun for their help and advice for my future endeavors, and to Nick Bambos and Olivier Gevaert for being on my defense committee. Finally, I want to thank Nectarios Koziris, my undergraduate advisor, for introducing me to research, and encouraging me to pursue graduate studies.

I have been truly lucky to interact with many brilliant people at Stanford. Thanks to all the members of the MAST group, past and current, for all I learned from them: Jacob Leverich, Daniel Sanchez, David Lo, Richard Yoo, Ana Klimovic, Grant Ayers, Mingyu Gao, Adam Belay, Felipe Munera, Camilo Moreno, Sam Grossman, Raghu Prabhakar, Rakesh Ramesh, Greg Kehoe, Tomer London, Asaf Cidon, Shingo Tanaka, Kenta Yasufuku, Woongki Baek, Michael Dalton, and Hari Kannan. Also thanks to George Michelogiannakis, John Brunhaver, Andrew Danowitz, Mario Flajslik, Krishna Malladi, and Nicole Rodia for all the technical, and non-technical discussions.

Special thanks to my officemates Grant and Richard for putting up with my occasional grumpiness, and our great cluster admins, Jacob and David for making sure my experiments would finish in time for each deadline. I also want to thank our wonderful admins Sue and Teresa for ensuring I never had to worry about anything administrative. Finally, I am grateful to Nick Arvanitidis and Facebook for supporting my Ph.D. studies financially through fellowships.

My years at Stanford have been made better by the many wonderful friends I met here and by the support of my friends back in Greece. I have certainly missed listing some below, but they can be identified as members of the Greek, Spanish, Belgian and other mafia organizations around the world. Thanks to George, Idoia, Mikel (and Naroa), Eleftheria, Peggy, Konstantinos, Gemma, Borja, Carlos, Yianis, Sotiria, Nicholas, Stephanie, Nadine, Olivier, Leen, Lynn, Adrian, Felix, Manu, George, Yorgos, Alexandros, Ioanna, Nicole, John, Sam, Andrew, Mario, Laura, TY, Kshipra, Cristina, Karthika, Diego, Daniel, and many others for the fun memories.

Finally, I want to thank Daniel, for his love, advice and for being there, even when “there” was in the other end of the world. Last but not least, my parents and family (especially my nephews and nieces) for always encouraging me to continue my studies and follow a career path I loved, and for “stoically” putting up with me being ten timezones away, although given the distance, they have certainly regretted the encouragement.

*“Have Ithaka always in your mind.
Your arrival there is what you are destined for.
But don’t in the least hurry the journey... ”
C.P. Cavafy*

Contents

Abstract	iv
Acknowledgements	vi
1 Introduction	1
1.1 Resource Efficiency Challenges	2
1.2 Contributions	3
1.3 Thesis Organization	8
2 Background & Motivation	9
2.1 Cloud Computing Background	9
2.2 Datacenter Scalability Challenges	11
2.3 Cluster Management	13
2.3.1 Understanding Resource Requirements	14
2.3.2 Resource Management Decisions	19
2.4 Data Mining in Systems	22
3 Paragon: QoS-Aware Scheduling in Heterogeneous Datacenters	24
3.1 Introduction	24
3.2 Fast & Accurate Classification	27
3.2.1 Collaborative Filtering Background	28
3.2.2 Classification for Heterogeneity	30
3.2.3 Classification for Interference	33
3.2.4 Putting It All Together	35

3.3	Paragon	36
3.3.1	Overview	36
3.3.2	Greedy Server Selection	37
3.3.3	Statistical Framework for Server Selection	39
3.3.4	Discussion	42
3.4	Methodology	43
3.5	Evaluation	46
3.5.1	Comparison of Schedulers: Small Scale	46
3.5.2	Comparison of Schedulers: Large Scale	52
3.6	Related Work	56
3.7	Conclusions	58
4	Quasar: QoS-Aware and Resource Efficient Cluster Management	60
4.1	Introduction	60
4.2	Motivation	65
4.2.1	Cluster Management Overview	65
4.2.2	The Case for Coordinated Cluster Management	66
4.3	Quasar	68
4.3.1	Overview	68
4.3.2	Fast and Accurate Classification	70
4.3.3	Greedy Allocation and Assignment	77
4.3.4	Putting it All Together	78
4.4	Implementation	80
4.4.1	Dynamic Adaptation	80
4.4.2	Side Effect Free Profiling	81
4.4.3	Stragglers	82
4.4.4	Discussion	82
4.5	Methodology	83
4.6	Evaluation	86
4.6.1	Single Batch Job	86
4.6.2	Multiple Batch Frameworks	88

4.6.3	Low-Latency Service	89
4.6.4	Stateful Latency-Critical Services	90
4.6.5	Large-Scale Cloud Provider	92
4.7	Conclusions	93
5	iBench: Quantifying Interference in Datacenter Workloads	95
5.1	Introduction	95
5.2	Related Work	98
5.3	iBench Workloads	99
5.3.1	Overview	99
5.3.2	Designing the SoIs	100
5.4	Validation	105
5.4.1	Individual SoIs Validation	105
5.4.2	SoI Impact on Applications	106
5.4.3	Correlation between SoIs	108
5.5	Use Cases	109
5.5.1	Datacenter Scheduling	109
5.5.2	Server Provisioning	112
5.5.3	Application Development/Testing	113
5.5.4	Scheduling in Heterogeneous CMPs	115
5.6	Conclusions	117
6	ARQ: QoS-Aware Admission Control	119
6.1	Introduction	119
6.2	Background	120
6.3	Admission Control	121
6.3.1	Overview	121
6.3.2	Waiting Time versus Resource Quality	123
6.4	Methodology	126
6.5	Evaluation	128
6.5.1	Small-scale Experiments	128
6.6	Related Work	131

6.7	Conclusions	133
7	Tarcil: Reconciling Scheduling Speed and Quality in Large Shared Clusters	134
7.1	Introduction	134
7.2	Background	138
7.3	The Tarcil Scheduler	140
7.3.1	Overview	140
7.3.2	Analytical Framework	140
7.3.3	Sampling-based Scheduling with Guarantees	143
7.4	Admission Control	146
7.4.1	Pre-scheduling Queueing	146
7.4.2	Post-scheduling Queueing	148
7.5	Tarcil Implementation	149
7.5.1	Tarcil Components	149
7.5.2	Adjusting Allocations	151
7.5.3	Fairness	152
7.6	Evaluation	153
7.6.1	Tarcil Analysis	153
7.6.2	Comparison with Other Schedulers	156
7.6.3	Large-Scale Evaluation	160
7.7	Conclusions	163
8	HCloud: Optimizing Resource Provisioning in Public Clouds	165
8.1	Introduction	165
8.2	Cloud Workloads and Systems	169
8.2.1	Workload Scenarios	169
8.2.2	Cloud Instances	170
8.2.3	Cloud Pricing	171
8.3	Provisioning Strategies	172
8.3.1	Statically Reserved Resources (SR)	172
8.3.2	Dynamic On-Demand Resources (OdF, OdM)	173

8.3.3	The Importance of Resource Preferences	174
8.3.4	Provisioning Strategies Comparison	176
8.4	Hybrid Provisioning Strategies	178
8.4.1	Provisioning Strategies	179
8.4.2	Application Mapping Policies	179
8.4.3	Provisioning Strategies Comparison	183
8.5	Discussion	185
8.5.1	Sensitivity to Job/System Parameters	185
8.5.2	Provisioning Overheads	188
8.5.3	Different Pricing Models	188
8.5.4	Resource Efficiency	190
8.5.5	Additional Provisioning Considerations	193
8.5.6	Sensitivity to Workload Characteristics	194
8.5.7	Cost Impact of Information from Quasar	195
8.6	Related Work	196
8.7	Conclusions	197
9	Conclusions and Future Work	199
A	Storage Modeling of Datacenter Workloads	201
A.1	Introduction	201
A.2	Related Work	203
A.3	Modeling and Generation Process	205
A.3.1	Basic State Diagram Model	205
A.3.2	Hierarchical State Diagram Model	206
A.3.3	Storage Activity Fluctuation	207
A.3.4	Generation Tool Design - DiskSpd	208
A.4	Characterization and Validation	212
A.4.1	Original DC Workloads	212
A.4.2	Generating Models from Traces	213
A.4.3	DC Application Characterization	213
A.4.4	Validation	215

A.4.5	Comparison with IOMeter	218
A.5	Use Cases	221
A.5.1	SSD caching	221
A.5.2	Defragmentation	222
A.6	Conclusions	224
B	BLOC: Bandwidth-Aware Storage Consolidation	225
B.1	Introduction	225
B.2	BLOC Design	228
B.2.1	System Overview	228
B.3	Quantifying Storage Interference	230
B.3.1	Motivation	230
B.3.2	Application Models	231
B.3.3	Trimming the Search Space	232
B.3.4	Profiling	232
B.3.5	Validation	233
B.4	Bin Packing	234
B.4.1	Limiting Storage Resource Usage	234
B.4.2	Bin Packing Algorithm	235
B.4.3	Consolidation Granularity	236
B.4.4	Dynamic Demands	237
B.5	Dynamic Adaptation	238
B.5.1	Predicting User Load	239
B.5.2	Dynamic Updates to Bin Packing	240
B.5.3	Migration Costs	240
B.5.4	Performance and Efficiency Trade-offs	242
B.6	Implementation	243
B.7	Evaluation	245
B.7.1	Comparison of Consolidation Schemes	246
B.7.2	BLOC Behavior	248
B.8	Discussion	250

B.9	Related Work	252
B.10	Conclusions	254
C	ECHO: Network Modeling of Datacenter Workloads	255
C.1	Introduction	255
C.2	Related Work	257
C.3	Single Server Temporal Model	259
C.4	Temporal and Spatial Network Traffic Characterization	261
C.5	System-Wide Spatial Model	265
	C.5.1 Overview	265
	C.5.2 Design	266
	C.5.3 Validation	271
C.6	Conclusions	281
	Bibliography	282

Chapter 1

Introduction

Cloud computing is at a critical junction. Its popularity has increased drastically over the past decade, with a growing number of applications and data hosted both on public and private clouds. At a high level, cloud computing offers three premises, both from the perspective of datacenter operators and end users: *flexibility* as resources can easily be obtained and released, *high performance*, and *cost efficiency*, as the infrastructure is shared across multiple users.

Despite its prevalence, cloud computing today faces significant scalability challenges. Traditionally, to scale a datacenter, operators would improve its cost efficiency or improve its compute capabilities. With respect to cost efficiency, two approaches have prevailed: switching from the specialized machines that used to populate these systems to commodity computing, and reducing the cost of power delivery and cooling. With most datacenters already consisting of commodity servers, and power delivery and cooling introducing less than 10% cost overheads, both approaches are reaching the point of diminishing returns. On the other hand, we can improve scalability by increasing the compute capabilities of a datacenter. This requires either building more datacenters, which requires a capital investment of hundreds of millions of dollars [31, 136], increasing the number of servers in each datacenter, albeit with bounded benefits, given the provisioning constraints of power delivery and cooling, or relying on the process technology to provide higher performance for the same power consumption. Unfortunately, process technology in microprocessors has slowed down

with the end of voltage (or Dennard) scaling, and with Moore’s Law projected to end in the next five to ten years, relying on hardware alone to improve datacenter scalability is not a viable option. This underlines the importance of operating datacenters at high utilization.

Unfortunately, while the large-scale datacenters that host cloud computing services have grown in number and size, the utilization at which they operate has remained prohibitively low [31, 79]. Even at the high end of the spectrum of production datacenters, utilizations rarely exceed 20-30% [31, 79], with most systems operating at even lower utilizations. This is the case even for systems that use virtualization [177] for multi-tenancy, and employ sophisticated systems to manage resources across applications. In the rest of this Section, we highlight the reasons behind low datacenter utilization, and the contributions of this thesis towards improving datacenter resource efficiency.

1.1 Resource Efficiency Challenges

At a high level, datacenter underutilization stems from two interacting factors. The first is the current *interface* between the users that submit applications to clusters and the system that must schedule these applications. Resources in datacenters today are allocated using a *reservation-based* API, where the user has to determine how many resources an application needs. Unfortunately this is extremely difficult for users, that tend to request a lot more resources than their applications truly require, resulting in grossly underutilized clusters.

The second reason behind datacenter underutilization justifies these exaggerated resource reservations. *Performance unpredictability* is the result of resource contention between applications sharing the system, platform heterogeneity as machines get progressively replaced over the provisioned lifetime of a datacenter, and fluctuations in user load. Unpredictability becomes even more of a challenge for user-interactive applications, like search, which have millions of users, experience spikes in their traffic and are provisioned for future growth. The performance metric of interest for such services is tail latency, which is much more difficult to satisfy than

average performance [73]. The risk of unpredictable performance together with the complexity of resource management leads users to significantly overprovisioning their resource reservations.

1.2 Contributions

The focus of this dissertation is to improve datacenter scalability, by increasing resource efficiency. While inefficiencies exist across the hardware and software stack, application performance and datacenter utilization are to a large extent determined by the cluster manager; the system that orchestrates resource allocation and application scheduling in large-scale systems. To this end, our contributions focus on increasing datacenter-wide utilization, by improving cluster management, while guaranteeing that each scheduled application satisfies its performance requirements.

To achieve this goal, we use three main insights. First, we take a *top-down approach* that bridges the different layers of the system stack from the user interface, to the cluster scheduler and down to hardware issues. Tackling the problem of datacenter efficiency cannot be addressed in a single level of the system stack, for example in hardware or software only. It requires in-depth understanding of the challenges and opportunities that each layer presents as well as their interactions. To this end, this work spans several levels of the stack from high-level distributed system design, to low-level architectural considerations.

Second, we introduce a *high-level, declarative interface* between users and system that focuses on performance, not resource reservations. We demonstrate that the existing reservation-based interface is poorly formalized and overly complex, leading to low system utilization. Instead, by simplifying the interface, the user is tasked with specifying *what* performance an application must achieve, not *how* to achieve it with low-level, raw resources.

Third, we show that the traditional, trial-and-error approach used in computer architecture and systems is prohibitively impractical at datacenter scale. We instead propose a new approach that leverages the knowledge on application behavior the system accumulates through data collection over time. By applying *data mining*

principles to these datasets in a *mindful fashion* we significantly improve both the quality and practicality of large-scale scheduling.

Using the insights described before, we have designed and built several systems, which together improve the performance and resource efficiency of cloud computing. Below, we provide a brief overview of each system.

Paragon: QoS-aware application assignment. Paragon is an online datacenter manager that, given a resource reservation, accounts for platform heterogeneity and workload interference in scheduling decisions [76, 80, 77, 78]. Paragon leverages two main insights.

First, Paragon takes into account the *various shared resources* where interference may occur, including the CPU, memory and I/O subsystems. To measure the sensitivity of an application to different sources of interference we designed iBench [74], a suite that consists of a set of benchmarks that put progressively more pressure on a specific resource, and can be used to determine how much interference a workload can tolerate in shared resources and how much pressure it itself creates.

Second, Paragon does not require detailed application profiling to extract its platform and interference preferences. Instead, it leverages the knowledge the system already has from previously-scheduled workloads. To this end, we designed an online recommender system, based on matrix factorization (SVD) and latent-factor models, that determines which platforms, and co-scheduled workloads will allow the job to satisfy its QoS constraints. The system is similar to the *recommender systems* used in sites like Netflix or e-commerce, where a sparse information signal for a new user is projected against the rich information from previous users to provide the new user with accurate movie or item recommendations.

Paragon is a practical system: the profiling and data mining techniques add minimal scheduling overheads. Using the information from the recommender system allows the scheduler to improve performance, and because applications are packed more tightly it also improves system utilization. In a 5,000 application scenario running on 1,000 servers on Amazon EC2, Paragon achieves performance within 4% of the optimal for that cluster. In comparison, a heterogeneity- and interference-agnostic scheduler degrades performance by 48% on average and violates QoS for

97% of workloads.

Quasar: Resource-efficient cluster management. Quasar takes the approach in Paragon one step further to address the more general problem of cluster management in datacenters [79]. Paragon has one limitation. While it can determine where to place an application in a large-scale system, it lacks the ability to determine how many resources that application needs to satisfy its performance constraints. This means that the user is still tasked with requesting an appropriate resource amount for a new job. However, because resource allocation is a complex, multi-dimensional problem, users rarely estimate their resource needs correctly. More often they over-provision their reservations, hurting system utilization. During a study of Twitter’s datacenters, we verified that reservations often exceed usage by an order of magnitude. Quasar addresses this issue through two main contributions.

First, Quasar shifts from the traditional reservation-based interface between user and system to a *high-level declarative interface*, which draws from the concept of Domain-Specific Languages (DSLs) and SQL. Now, instead of the user specifying the low-level, raw resources (e.g., memory, cores, storage) he expects an application to need, he simply specifies the performance target the new application must meet, for example tail latency. This simplifies the responsibility of the user, and gives enough flexibility to the cluster manager to better place jobs on available resources. Subsequently the cluster manager translates this performance goal to resources.

To perform this translation, Quasar leverages *fast data mining techniques* in a similar way to Paragon. The difference is that the system must also provide recommendations on the amount of resources a job needs, specifically the resources per node and the number of nodes across which the workload should be distributed. At first glance this adds significant complexity to the recommender system, posing the question: can we maintain all this information, but solve a much simpler problem? We address this question by decomposing the problem to the four dimensions of resource allocation that affect application performance: platform heterogeneity, workload interference, scale-up (resources per node) and scale-out (number of nodes). This dramatically reduces the scheduling overheads, without sacrificing scheduling quality. Importantly, Quasar enables common datacenter application functionality, including

both distributed batch workloads and user-interactive services that are more sensitive to unpredictability.

Quasar is practical. The information required for the cluster manager’s decisions adds negligible overheads to application execution. These overheads can be further reduced as more sophisticated data mining techniques are designed. In a 200-server EC2 cluster Quasar meets the QoS requirements of 95% out of 1,200 applications, while increasing system utilization by more than 2x. In contrast, a traditional reservation-based cluster manager with a baseline least-loaded scheduler violates QoS for most workloads, despite system utilization rarely exceeding 25%. The approach proposed with Quasar has had some early adoption in real production systems, with both Twitter and AT&T adopting similar approaches in their latest system designs.

Tarcil: Improving scheduling scalability. The structure of the scheduler itself and the algorithms it employs are critical components of a cluster manager. Traditionally there has been a disparity in cluster scheduling. On one hand there are sophisticated, centralized schedulers, that examine all (or most) of the cluster state to improve scheduling quality (e.g., Quasar). On the other hand there are distributed, typically sampling-based, schedulers that make fast decisions by only examining a small subset of resources. Unfortunately, neither approach achieves both high scheduling quality and high scheduling speed. To bridge this gap we developed Tarcil [81, 82], a cluster scheduler designed both for short and long jobs. Tarcil is built on two insights: first, it accounts for the resource preferences of incoming workloads, to keep scheduling quality high. Second, it uses adaptive resource sampling to reduce the latency of each scheduling decision. The sample size is set based on how strictly QoS must be met, following a simple analytical framework that provides statistical guarantees on scheduling quality. Tarcil also uses admission control that takes action when load is very high to avoid overloading the system. Admission control determines how long applications should be queued for before being scheduled [72]. Finally, Tarcil is structured as a distributed system with multiple concurrent scheduling agents making task placement decisions.

In EC2 clusters with several hundred machines, Tarcil improves performance by 41% on average for short tasks over state-of-the-art distributed schedulers, and

scheduling latency by 1-2 orders of magnitude compared to sophisticated, centralized systems.

Cost-efficient cloud provisioning strategies. So far we assume that the cluster manager has full control over the entire system. In practice, many systems are deployed in public cloud providers where the visibility an external scheduler has is limited. In this case, apart from deciding the amount of needed resources, the user must also decide between *on-demand*, *reserved* and *spot instances*, each of which has different advantages and challenges. In this work, we present a set of provisioning techniques that, based on the characteristics of incoming workloads, determine the most appropriate type and size of instances that should be purchased. They also determine how long resources should be retained for once idle, to prevent re-instantiation overheads during periods of high load.

We have evaluated the system on a large Google Compute Engine cluster with several hundred instances, and showed that a hybrid configuration with both reserved and on-demand instances improves performance by $2.1\times$ over a fully on-demand system, and reduces cost by 46% over a fully-reserved system [75].

Other contributions: Finally, a major roadblock in datacenter research in academia is the lack of representative, open-source datacenter applications. To address this issue, we designed analytically-driven models that capture the temporal and spatial characteristics of datacenter application storage and network activity and can generate representative access patterns [83, 84, 85, 86, 87, 88]. These models are validated against real datacenter applications running in Microsoft’s cloud facilities and used for troubleshooting and to identify hardware and software inefficiencies, such as imbalanced data sharding and suboptimal SSD caching without the need for a full system deployment.

1.3 Thesis Organization

The rest of this thesis is organized as follows. Chapter 2 provides relevant background and motivation. Chapter 3 presents Paragon, a QoS-aware datacenter scheduler that accounts for platform heterogeneity and application interference. Chapter 4 discusses Quasar, a new cluster manager that introduces a high-level declarative interface between users and system, and leverages data mining to provide efficient, and high-quality resource allocations. In Chapter 5 we present iBench, a benchmark suite that quantifies the impact of interference in the various shared system resources. In Chapter 6 we describe ARQ, a multi-class admission control protocol that prevents the system from becoming oversubscribed. Chapter 7 discusses Tarcil, a scalable and low-latency sampling-based datacenter scheduler, and Chapter 8 presents resource provisioning strategies in the presence of hybrid resources, i.e., when part of the system resides on a public cloud provider. Finally, Chapter 9 concludes this dissertation. Appendices A and B discuss an analytical model for the storage activity of datacenter workloads, and a bandwidth-aware storage consolidation system that leverages this model. Appendix C presents a similar analytical approach for the network activity of distributed datacenter applications.

Chapter 2

Background & Motivation

2.1 Cloud Computing Background

Datacenters, the large-scale infrastructures that host cloud computing services, have experienced a rapid increase in both their number and size in the past ten years [31]. Cloud computing has become an essential tool and a catalyst for innovation in all aspects of endeavor, including healthcare, education and science [217]. Both private and public datacenters with tens of thousand of machines now host popular services, such as search, social networking, email, video streaming, enterprise management tools, maps, natural language processing, big-data analytics and general purpose storage platforms [13, 279, 45, 218, 110, 65]. We have come to expect that these services provide us with real-time, personalized, and contextual access to terabytes of data.

The popularity of cloud computing services has led to significant work on analyzing these workloads and the infrastructure that supports them. Researchers have characterized the behavior of webserving environments [5, 42, 216], search engines [30, 220], distributed computing frameworks like MapReduce [57, 157, 69], memory-based storage services [23, 70], and large-scale storage systems [9, 156, 129, 85]. Recent studies have used datacenter traces to extract observations on the duration, CPU and memory usage, and variability of cloud workloads [222, 89, 223, 179, 295]. These observations are particularly important in guiding scheduling decisions. Finally, researchers have

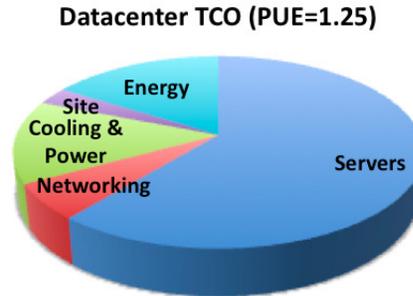


Figure 2.1: (a) Breakdown of the total cost of ownership (TCO) of a 10MW datacenter assuming 3 year server amortization and 15 year facility amortization [262, 136].

developed guidelines on how to study and benchmark these workloads at scale [233].

Datacenter services fall in one of two main categories; *batch analytics* workloads [69, 288], and *latency-critical* online services [30, 107, 49, 51]. Analytics optimize for computation throughput as they process vast amounts of data, for example user preferences for advertisements or movies. Online services, on the other hand, are user-interactive applications, such as search and email, and they must meet strict latency (or response time) constraints. Additionally, due to the organization of these distributed services, latency requirements are formalized with respect to tail latency, for example 99th percentile, instead of average response time [68]. This puts increased pressure on the system to behave in a high-performant and predictable way.

In general, cloud computing offers three main premises, both to end users and datacenter operators; *resource flexibility*, *high performance*, and *cost efficiency*. Users can increase or decrease the resources they use at runtime according to the needs of their applications, and only pay for resources used at each point in time. Additionally, hosting applications in the cloud is less costly for users than setting up and maintaining a local infrastructure, even for large-scale services that require thousands of machines, such as Netflix [202]. At the same time, datacenter operators achieve cost benefits by sharing their infrastructure across multiple tenants.

The primary cost metric in datacenters is the total cost of ownership (TCO). The TCO includes both the capital expenditures to build a datacenter and populate it with servers (CAPEX), and its operational expenses in terms of power consumption,

cooling and maintenance (OPEX). Figure 2.1 shows a breakdown of the TCO of a large-scale datacenter [136]. The capital expenses for purchasing the servers accounts for 61% of the total cost of the system, while the energy to power the machines amounts to another 18%. This places a particular emphasis on how well datacenter resources are being used. The next section details the scalability challenges that stem from poor datacenter utilization.

2.2 Datacenter Scalability Challenges

In the past ten years, operators have scaled the capabilities of cloud services by building larger datacenters that can host tens to hundreds of thousands of multi-core servers [137]. The servers are connected by networks with high-speed links (e.g., 10Gbps Ethernet) and advanced topologies that support high bandwidth between any two servers [11, 122]. At the same time, operators leveraged two approaches to improve cost efficiency. First, they switched from the specialized machines that used to populate datacenters to commodity servers that benefit from economies of scale. Second, they reduced the cost and energy overheads of the power delivery and cooling infrastructure [137]. While a few years ago the power usage effectiveness (PUE) of datacenters was as high as 3.0, the PUE of modern facilities is as low as 1.1, reaching the point of diminishing returns¹.

Unfortunately, we have reached the end of the road for these scaling techniques. Datacenters are already consuming tens of MWatts, stressing the capabilities of power generation facilities and making it difficult to continuously increase the number of servers per facility [137, 262]. At the same time, the end of voltage (or Dennard) scaling, and the projected end of Moore’s Law mean that hardware alone can no longer provide improved performance for the same power budget [142, 96, 158, 64].

To achieve further improvements in datacenter scalability, we must improve their

¹PUE of 3.0 indicates that for every 1W consumed by the servers, another 2W are consumed by the power delivery and cooling infrastructure. PUE of 1.1 indicates that the overhead of power delivery and cooling is merely 10%.

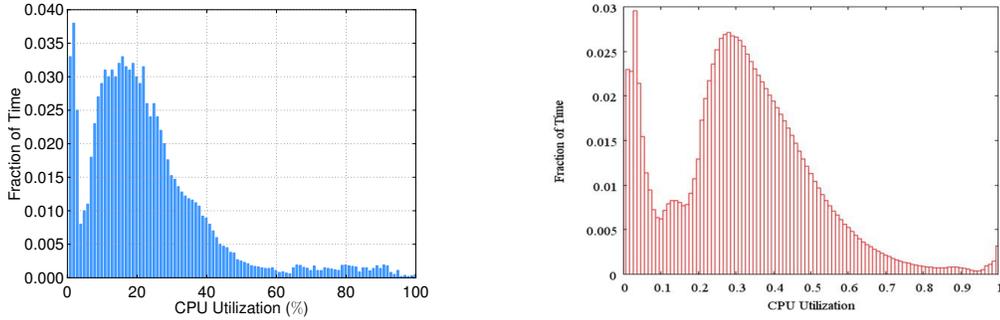


Figure 2.2: Probability distribution functions (PDF) of the CPU utilization in a production datacenter at Twitter (left) [79] and Google (right) [31].

resource efficiency, i.e., we must extract as much compute as possible from the resources available in these systems today [29]. Figure 2.2 shows the probability distribution function of CPU utilization in a production datacenter at Twitter (left) and Google (right) [79, 31]. Both systems consist of several thousand machines servicing user-interactive and analytics jobs, and enable resource sharing across applications with techniques like containerization and virtualization. Nevertheless, utilization is quite low, typically ranging from 10% to 30% of the system’s nominal compute capabilities, and wasting a significant fraction of capital expenses invested towards purchasing the server infrastructure. Hence the obvious path forward is to increase the utilization of datacenter servers. High server utilization is also beneficial for energy efficiency. Since most servers are not energy proportional, consuming 40% to 60% of their peak power when idling [32, 29, 31, 170, 191], they operate more efficiently at high utilization.

Nevertheless, there are several challenges towards achieving resource efficiency through high server utilization.

First, datacenter operators must plan for diurnal usage patterns, unexpected spikes in user demands, and future growth. This results in significantly overprovisioned resource allocations, leaving servers underutilized for most of the time.

Second, datacenters are inherently heterogeneous, both in an effort to match the requirements of widely diverse applications, and due to the progressive server replacement during the typical 15-year lifetime of a datacenter infrastructure [31, 136, 137,

163, 186, 199]. At any point in time, a datacenter may host 3-5 server generations with a few hardware configurations per generation, in terms of the specific speeds and capacities of the processor, memory, storage and networking subsystems. Hence, it is common to have 10 to 40 configurations throughout the datacenter. Ignoring heterogeneity can lead to significant inefficiencies, as some workloads are sensitive to the hardware configuration.

Third, and most important, increasing server utilization by scheduling multiple services on each server leads to *performance loss due to interference*. Even when using different processor cores, co-scheduled applications can interfere on shared caches, memory channels, storage and networking devices [120, 188, 200].

Interference is particularly detrimental for *latency-critical, user-facing services with strict quality-of-service (QoS) guarantees*. For instance, updating a social networking news feed involves queries for the user’s connections and recent status updates; ranking, filtering, and formatting updates; retrieving related media files; and selecting and formatting relevant advertisements and recommendations. Since tens of servers are involved in each user query, low average latency is not sufficient. The requirement is for low tail latency (e.g., low 95th or 99th percentile) so that latency variability does not impact a significant percentage of user requests. Assigning additional workloads to each server to raise utilization typically leads to higher latency and higher variability. Latency-critical requests may be queued for milliseconds if other tasks occupy processing cores. But even if cores are available, the latency-critical requests may underperform due to interference and contention on shared resources. Hence, it is common for latency-critical services to be deployed on dedicated machines, which are underutilized for the majority of time.

2.3 Cluster Management

Datacenters are commonly orchestrated by systems called *cluster managers*. Cluster managers are primarily responsible for scheduling incoming applications and managing system resources [232, 139, 267]. They also have additional responsibilities that pertain to fault tolerance [174, 115, 139], reliability [259, 299], enforcement of security

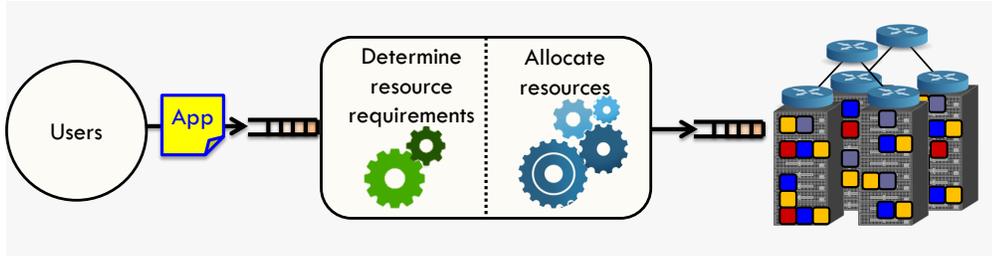


Figure 2.3: Cluster manager structure.

constraints [146], and various monitoring capabilities [62, 139]. In the context of this dissertation we focus on the responsibilities of cluster managers with respect to the management of applications and system resources.

Figure 2.3 shows a simplified overview of the functionality of a cluster manager with respect to managing system resources. This functionality can be divided to two main components: first, the system must *understand the resource requirements* incoming, potentially-unknown applications have. This includes determining both the amount of resources an application needs, and the specific configuration of resources required. Second, the system must *make and enforce resource allocation decisions* in a way that satisfies the resource requirements of incoming jobs. For both components there is a wide spectrum of designs and implementations. The following sections provide an overview of related work in each area.

2.3.1 Understanding Resource Requirements

Cloud services are widely diverse with respect to their resource requirements. While certain applications, such as websearch, require the latest platforms to deliver low end-to-end request latencies, other services, such as background analytics and logging operations are less sensitive to allocated resources. Current cluster management interfaces are *reservation-based*, tasking users with specifying the resources a new, potentially-unknown job should receive. However, determining the appropriate resources for a new application is a very complex, multidimensional problem. Consider, for example, a single service whose required resources we want to establish. For the

purpose of the example, we select *memcached*, a low-latency, in-memory key-value store [107]. We first need to determine how the performance for memcached scales as we increase the number of cores allocated to the application in a single node (*scale-up*). For simplicity memory is unconstrained. We are interested in the maximum throughput, in queries per second (QPS) that memcached can achieve, such that its 99th percentile latency is below 200usec. Figure 2.4a shows this scaling experiment. Since memcached is memory-bound, increasing the number of cores beyond a certain point does not benefit performance. The experiment is conducted on a 12 core Intel Xeon server (CPU E5-2630 @ 2.30GHz) with 64GB of RAM. Since datacenters consist of several hardware platforms, we need to repeat this experiment on each of them. Figure 2.4b shows the different scaling curves for servers ranging from low-end Clovertown-based servers to high-end Haswell-based machines. Performance varies widely, as the low-end platforms become saturated much faster. In addition to the type of platform, a user must also know how the performance of a job scales as the load becomes distributed across an increasing number of servers (*scale-out*). Figure 2.4c shows the throughput of memcached in QPS when scaling out from a single Xeon server to 8 servers of the same type. Since there is limited communication between servers, performance is almost linear for memcached. This, however, is not the case in general, especially for applications sharing common state.

So far we have maintained the characteristics of the input load constant, i.e., same key-value distributions. In real datacenter settings load fluctuates widely as user traffic is higher during the day, and drops during the night. This is mostly the case for user-interactive services, such as search, email, and social networks. Batch applications, like analytics, also experience variations in the size and characteristics of their input datasets. Figure 2.4d shows how performance changes, as we vary the read:write request ratio and the size of keys and values. As expected when requests are write-dominated (*L3*), the throughput is significantly lower.

Finally, a principal objective of the cluster manager is to increase system utilization, by sharing resources across applications. Unfortunately, resource sharing incurs interference due to contention. Understanding the sensitivity of an application to different types and intensities of interference is a critical dimension in cluster

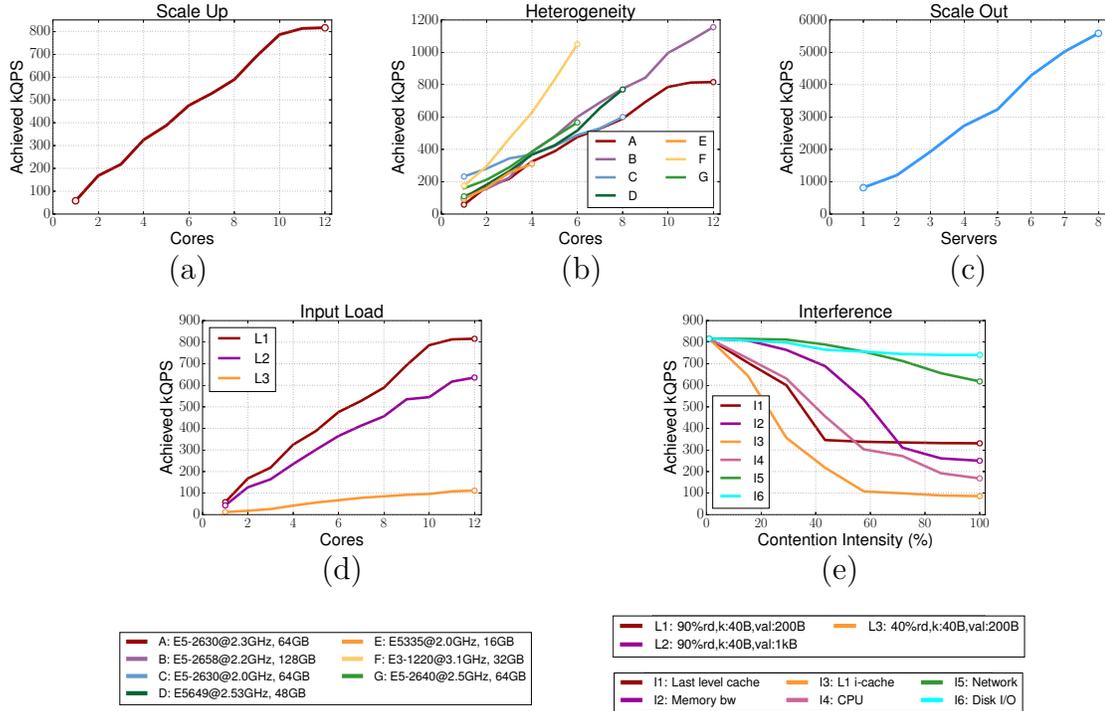


Figure 2.4: The steps needed to determine the appropriate resources for a new, unknown workload.

management. Figure 2.4e shows how the performance of memcached changes with increasing amounts of interference in various resources. Moreover, since most of these dimensions are affected by the application itself and the system setup, the exploration must be repeated upon application updates, operating system upgrades, or the introduction of new hardware platforms.

This analysis highlights the fact that understanding the resource requirements of a potentially new application is a challenging problem. This is, however, the task that most cluster management frameworks require users to perform. These interfaces foster underutilization, as conservative users overprovision their reservations to avoid insufficient resources, and performance unpredictability.

There has been significant work on determining the *amount* of resources needed by an application in both virtualized and non-virtualized systems [50, 54, 61, 93, 99, 118, 119, 124, 238, 248, 261, 276, 298, 300]. Rightscale [224], for example, uses a load threshold to automatically scale out 3-tier applications to react to changes in

the load in Amazon’s cloud service [13]. CloudScale identifies application resource requirements using online demand prediction and prediction error handling, without a priori assumptions on application behavior [240]. Dejavu serves a similar goal by identifying a few workload classes and based on them, reuses previous resource allocations to minimize reallocation overheads [266]. Dominant Resource Fairness (DRF) [114] provides a generalization of max-min fairness across multiple resources that disincentivizes users from lying about their requirements, and preventing resource sharing. Zhu et al. [300] present a resource management scheme for virtualized datacenters that preserves service level agreements (SLAs), and Gmach et al. [117] present a resource allocation scheme for datacenter applications that relies on the ability to predict their behavior a priori. Finally, there is a lot of work on determining allocation requirements in virtualized environments [3, 284, 249, 250], and reclaiming unused resources that can service new load [1, 48, 274, 272].

Resource assignment, i.e., determining the appropriate *type* of allocated resources, is equally critical to resource sizing. Given the fact that platforms vary considerably in modern datacenters, and that interference in shared resources is detrimental to performance, understanding how application behavior changes with *platform heterogeneity* and *workload interference* is essential.

Recent work on datacenter management has highlighted the importance of these two factors. Mars et al. [186, 187] have shown that the performance of Google workloads can vary by up to 40% due to heterogeneity even when considering only two server configurations and up to 2x due to interference even when considering only two colocated applications. In [186], they present a system that uses combinatorial optimization to select the proper server configuration for a given workload. In [187], they present a two-step method to characterize the sensitivity of workloads to memory pressure and the stress each application exercises to the memory subsystem. In the same spirit Yang et al. [285] apply a dynamic interference sensitivity detection scheme to preserve the performance of batch and latency-critical applications under collocation scenarios. Govindan et al. [120] also present a scheme to quantify the effects of cache interference between consolidated workloads, although they require access to physical memory addresses. Zhang et al. [296] use cycles-per-instruction (CPI)

as a proxy for interference between workloads and throttle the offending co-runners such that the applications return to their expected behavior. Quincy [149] formulates resource assignment as a graph optimization problem, accounting for fairness, and placement constraints application may have. Finally, Nathuji et al. [200] present a control-based resource allocation scheme that mitigates the effects of cache, memory and hardware prefetching interference between co-scheduled workloads. While these systems highlight the importance of factoring heterogeneity and interference in scheduling decisions, they incur significant profiling overheads, and are limited to capturing interference in a small number of shared resources.

The problem of resource assignment is well-established in systems using virtualization. VM management systems such as vSphere [273], XenServer [4] or the VM platforms on EC2 [13] and Windows Azure [279] can schedule diverse workloads submitted by a large number of users on the available servers. In general, these platforms account for application resource requirements which they learn over time by monitoring workload execution. VMWare’s Distributed Resources Scheduler (DRS) [271] for example accounts for CPU and memory requirements when scheduling applications. Recently, DeepDive [204] proposed a black-box system for management of virtual machines which accounts for interference, while minimizing migration overheads.

Finally, resource management in heterogeneous CMPs shares some concepts and challenges with datacenter management. Fedorova et al. [102] discuss OS level scheduling for heterogeneous multi-cores as having the following three objectives: optimal performance, core assignment balance and response time fairness. Shelepov et al. [239] present a resource manager that exhibits some of these features and is simple and scalable, while Craeynest et al. [263] use performance statistics to estimate which workload-to-core mapping is likely to provide the best performance. Datacenter management also has similar requirements as applications should observe their QoS, resource allocation should follow application requirements closely and fairness between co-scheduled workloads should be preserved.

2.3.2 Resource Management Decisions

Once the cluster manager has been given, or has determined itself, the resource requirements of a new application, it must perform a scheduling decision. Ideally, datacenter management should have three desirable properties. First, each workload should receive the resources that enable it to achieve *high and predictable performance*. Second, jobs should be tightly packed on available servers to achieve *high cluster utilization*. Third, scheduling overheads should be minimal to allow the scheduler to *scale to large clusters and high job arrival rates*. With these three objectives in mind, cluster schedulers follow a diverse set of designs.

Cluster managers can be examined along two dimensions with respect to their scheduling decisions: *scheduling concurrency (throughput)* and *scheduling speed (latency)*.

With respect to scheduling concurrency, there are two groups of work. In the first, scheduling is serialized, with a centralized scheduler making all decisions [149, 79]. However, application scheduling in clusters with thousands of servers and high workload churn becomes a bottleneck. The second group of work addresses this problem by scheduling multiple jobs in parallel through two-level, distributed or shared-state designs [139, 232]. Two-level schedulers, such as Mesos and YARN, use a centralized coordinator to divide resources between frameworks like Hadoop, Spark and MPI [139, 267]. Each framework uses its own scheduler to assign resources to tasks. Since neither the coordinator nor the framework schedulers have a complete view of the cluster state and all task characteristics, scheduling is suboptimal [232]. Shared-state schedulers like Omega [232] allow multiple scheduling agents to concurrently access the whole cluster state using atomic transactions. As long as these agents rarely attempt to schedule work to the same servers (infrequent conflicts), concurrency comes with a low performance cost. Finally, Sparrow uses multiple concurrent, stateless schedulers to sample and allocate resources [209].

With respect to the speed at which scheduling decisions happen, there are again two groups of work. The first group examines most of (or all) the cluster state to determine the most suitable resources for incoming tasks, in a way that addresses

the performance impact of *hardware heterogeneity* and *interference in shared resources* [76, 120, 285, 200, 186, 296, 241]. For instance, Quincy [149] formulates scheduling as a cost optimization problem that accounts for job preferences with respect to locality, fairness and starvation-freedom. Similarly, Tetris [121] uses a greedy algorithm to pack machines in a way that matches the resource requirements of tasks to the resource availability of a particular machine. These schedulers make high-quality decisions that lead to high application performance and high cluster utilization. However, they inspect the full cluster state on every scheduling event. Their decision overhead can be prohibitively high for large clusters, and in particular for the very short jobs of real-time analytics (100 ms–10 s) [209, 288]. Using multiple greedy schedulers improves scheduling throughput but not latency, and terminating the greedy search early hurts decision quality, especially at high cluster loads.

The second group leverages results from randomized load balancing [193, 212], to design sampling-based cluster schedulers [52, 91, 209]. Sampling the state of just a few servers reduces the latency of each scheduling decision and the probability of conflicts between concurrent agents, and is likely to find available resources in non heavily-loaded clusters. The recently-proposed Sparrow scheduler uses *batch sampling* and *late binding* [209]. Batch sampling examines the state of two servers for each of m required cores by a new job and selects the m best cores. If the selected cores are busy, tasks are queued locally in the sampled servers and assigned to the machine where resources become available first. While sampling-based schedulers improve scheduling speed, their decisions can be poor because they ignore the resource preferences of jobs. Typically concurrent schedulers follow sampling schemes, while centralized systems are paired with sophisticated algorithms. In Section 7 we present a cluster scheduler that bridges the disparity between the high quality and low speed centralized schedulers and the high speed and low quality of distributed, sampling-based systems.

Public clouds becoming the platform of choice for many cloud services users has also motivated a large body of work on optimizing cloud provisioning for both performance and cost. For example, Deelman et al. [71] discuss cost-efficient provisioning strategies for specific astronomy applications on a cloud provider. Li et al. [172]

compare the resource pricing of several cloud providers to help users provision their applications. There are also studies that analyze resource pricing strategies in public clouds, and contest whether pricing is indeed market-driven, for example for spot instances on Amazon EC2, compared to alternative strategies [36]. Finally, Guevara et al. [127] and Zahed et al. [290] have incorporated the economics of heterogeneous resources in market-driven and game-theoretic strategies for resource allocation in shared environments.

While cloud providers are suitable for several online services, there are others that due to security and privacy concerns or cost limitations are still hosted on private systems. Trying to achieve the best of both worlds, many cloud computing users now deploy *hybrid clouds*, which consist of both privately-owned and publicly-rented machines [20, 44, 143, 159, 294]. Hybrid clouds raise additional provisioning challenges, as a user must now determine not only the type and configuration of resources rented on a public cloud, but in addition how to partition the load (and data) between private and public machines. Breiter et al. [44], for example, have described a framework that allows service integration in hybrid cloud environments, including actions such as overflowing in on-demand resources during periods of high load. Farahabady et al. [143] also present a resource allocation strategy for hybrid clouds that attempts to predict the execution times of incoming jobs and based on these predictions generate Pareto-optimal resource allocations. Finally, Annapureddy et al. [20] and Zhang et al. [294] discuss the security challenges of hybrid environments, and propose ways to leverage the private portion of the infrastructure for privacy-critical computation. In such settings, where the options for resource offerings are plentiful, understanding the requirements of scheduled applications becomes even more critical. In Section 8 we show that by accounting for the resource preferences of incoming workloads, a hybrid provisioning system can improve over the performance of public resources, and the cost efficiency of private, reserved servers.

2.4 Data Mining in Systems

The conventional design approach in architecture and systems has several drawbacks for large-scale datacenters. For example, while in a traditional desktop or mobile system, exhaustively characterizing the behavior of the handful of applications of interest would be a viable solution, the scale at which datacenters operate do not allow for such *best-effort* designs. Specifically, because instead of a few cores or servers, we now have tens to hundreds of thousands of machines running diverse applications with a high churn, we need *practical solutions*, that quickly and accurately determine the resource requirements of new workloads, and can provide guarantees on performance and system efficiency. Unfortunately, the empirical approach adopted so far cannot provide such practical designs, and instead results in overly complex solutions with poor predictability, leading to overprovisioning and underutilization.

A major contribution of this thesis is the introduction of a new approach in solving large-scale systems problems that relies on data mining. While machine learning techniques have been previously applied in system management [135, 152, 278, 41, 190], they were designed for small-scale systems, making their computational overheads when scaling to the hundreds of thousands of servers in modern datacenters impractical. Instead, in this dissertation we focus on simple data mining techniques that leverage the massive amounts of monitoring data, including information on the behavior of scheduled applications, datacenters collect today. We show that by *mining* this data in a *mindful fashion* we can not only get rich insights on the resource requirements of previously-unseen applications, but also produce practical solutions for cluster management that can be deployed in real-world environments and benefit both performance and resource efficiency. Specifically, with Paragon we show that data mining can help the scheduler determine which hardware platform is most suitable for a given workload, as well as the sensitivity an application has to different types of interference. With Quasar we generalize this insight to solve the more general cluster management problem, where the system must also determine the amount of resources needed by an application, without burdening the user with specifying

resource reservations. This enables not only high and predictable application performance, but allows datacenters to operate at 2-3x higher utilizations than before.

Chapter 3

Paragon: QoS-Aware Scheduling in Heterogeneous Datacenters

3.1 Introduction

An increasing amount of computing is performed in the cloud, primarily due to cost benefits for both the end-users and the operators of datacenters (DC) that host cloud services [31]. Large-scale providers such as Amazon EC2 [13], Microsoft Windows Azure [279], Rackspace [218] and Google Compute Engine [110] host tens of thousands of applications on a daily basis. Several companies also organize their IT infrastructure as private clouds, using management systems such as VMware vSphere [273] or Citrix XenServer [4].

The operator of a cloud service must schedule the stream of incoming applications on available servers in a manner that achieves both *fast execution* (user’s goal) and high *resource efficiency* (operator’s goal), enabling better scaling at low cost. This scheduling problem is particularly difficult as cloud services must accommodate a diverse set of workloads in terms of resource and performance requirements [31]. Moreover, the operator often has no a priori knowledge of workload characteristics.

In this chapter, we focus on two basic challenges that complicate scheduling in large-scale DCs: *hardware platform heterogeneity* and *workload interference*.

Heterogeneity occurs because servers are gradually provisioned and replaced over

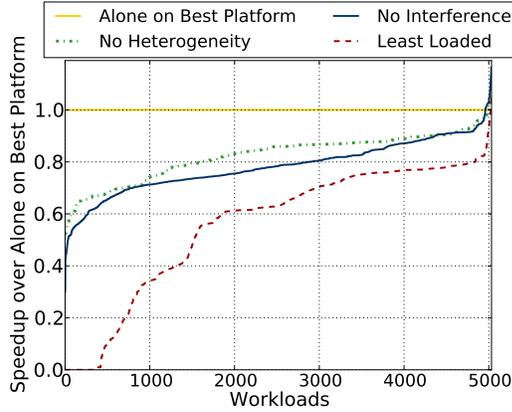


Figure 3.1: Performance degradation for 5,000 applications on 1000 EC2 servers with heterogeneity-oblivious, interference-oblivious and baseline least-loaded schedulers compared to ideal scheduling (application runs alone on best platform). Results are ordered from worst to best-performing workload.

the typical 15-year lifetime of a DC [31, 136, 163, 188, 199]. At any point in time, a DC may host 3-5 server generations with a few hardware configurations per generation, in terms of the specific speeds and capacities of the processor, memory, storage and networking subsystems. Hence, it is common to have 10 to 40 configurations throughout the DC. Ignoring heterogeneity can lead to significant inefficiencies, as some workloads are sensitive to hardware configurations. Figure 3.1 shows that a heterogeneity-oblivious scheduler will slow applications down by 22% on average, with some running nearly 2x slower (see Section 3.4 for methodology). This is not only suboptimal from the user’s perspective, but also for the DC operator as workloads occupy servers for significantly longer.

Interference is the result of co-scheduling multiple workloads on a single server to increase utilization and achieve better cost efficiency. By co-locating applications a given number of servers can host a larger set of workloads (better scalability). Alternatively, by packing workloads in a small number of servers when the overall load is low, the rest of the servers can be turned off to save energy. The latter is needed because modern servers are not energy-proportional and consume a large fraction of peak power even at low utilization [29, 31, 171, 191]. Co-scheduled applications may interfere negatively even if they run on different processor cores because they share

caches, memory channels, storage and networking devices [120, 187, 200]. Figure 3.1 shows that an interference-oblivious scheduler will slow workloads down by 34% on average, with some running more than 2x slower. Again, this is undesirable for both users and operators. Finally, a baseline scheduler that is both interference and heterogeneity-oblivious and schedules applications to least-loaded servers is even worse (48% average slowdown), causing some workloads to crash due to resource exhaustion on the server.

Previous work has showcased the potential of heterogeneity and interference-aware scheduling [188, 187]. However, techniques that rely on detailed application characterization cannot scale to large DCs that receive tens of thousands of potentially unknown workloads every day [45]. Most cloud management systems have some notion of contention or interference-awareness [139, 200, 266, 2, 283, 4]. However, they either use empirical rules for interference management or assume long-running workloads (e.g., online services), whose repeated behavior can be progressively modeled. In this work, we target both heterogeneity and interference and assume no a priori analysis of the application. Instead, we leverage information the system *already* has about the large number of applications it has previously seen.

We present *Paragon*, an online and scalable datacenter scheduler that is heterogeneity and interference-aware. The key feature of Paragon is its ability to quickly and accurately classify an unknown application with respect to heterogeneity (which server configurations it will perform best on) and interference (how much interference it will cause to co-scheduled applications and how much interference it can tolerate itself in multiple shared resources). Paragon’s classification engine exploits existing data from previously scheduled applications and offline training and requires only a minimal signal about a new workload. Specifically, it is organized as a low-overhead recommendation system similar to the one deployed for the Netflix Challenge [34], but instead of discovering similarities in users’ movie preferences, it finds similarities in applications’ preferences with respect to heterogeneity and interference. It uses singular value decomposition to perform collaborative filtering and identify similarities between incoming and previously scheduled workloads.

Once an incoming application is classified, a greedy scheduler assigns it to the

server that is the best possible match in terms of platform and minimum negative interference between all co-scheduled workloads. Even though the final step is greedy, the high accuracy of classification leads to schedules that satisfy both user requirements (fast execution time) and operator requirements (efficient resource use). Moreover, since classification is based on robust analytical methods and not merely empirical observation, we have strong guarantees on its accuracy and strict bounds on its overheads. Paragon scales to systems with tens of thousands of servers and tens of configurations, running large numbers of previously unknown workloads.

We implemented Paragon and evaluated its efficiency using a wide spectrum of workload scenarios (light, high, and oversubscribed). We use Paragon to schedule applications on a private cluster with 40 servers of 10 different configurations and on 1000 exclusive servers on Amazon EC2 with 14 configurations. We compare Paragon to a heterogeneity-oblivious, an interference-oblivious and a state-of-the-art least-loaded scheduler, which ignores both heterogeneity and interference. For the 1000-server experiments and a scenario with 2500 workloads, Paragon maintains QoS for 91% of workloads (within 5% of their performance running alone on the best server). The heterogeneity-oblivious, interference-oblivious and least-loaded schedulers offer such QoS guarantees for only 14%, 11%, and 3% of applications respectively. The results are more striking in the case of an oversubscribed workload scenario, where efficient resource use is even more critical. Paragon provides QoS guarantees for 52% of workloads and bounds the performance degradation to less than 10% for an additional 33% of workloads. In contrast, the least-loaded scheduler dramatically degrades performance for 99.9% of applications. We also evaluate Paragon on a Windows Azure and a Google Compute Engine cluster and show similar gains. Finally, we validate that Paragon’s classification engine achieves the accuracy and bounds predicted by the analytical methods and evaluate various parameters of the system.

3.2 Fast & Accurate Classification

The key requirement for heterogeneity and interference-aware scheduling is to quickly and accurately classify incoming applications. First, we need to know how fast an

application will run on each of the tens of server configurations available. Second, we need to know how much interference it can tolerate from other workloads in each of several shared resources without significant performance loss and how much interference it will generate itself. Our goal is to perform online scheduling for large-scale DCs without any a priori knowledge about incoming applications. Most previous schemes address this issue with detailed but offline application characterization or long-term monitoring and modeling approaches [187, 200, 266]. Instead, Paragon takes a different perspective. Its core idea is that, instead of learning each new workload in detail, the system leverages information it already has about applications it has seen to express the new workload as a combination of known applications. For this purpose we use collaborative filtering techniques that combine a minimal profiling signal about the new application (e.g., a minute’s worth of profiling data on two servers) with the large amount of data available from previously scheduled applications. The result is fast and highly accurate classification of incoming applications with respect to both heterogeneity and interference. Within a minute of its arrival, an incoming workload can be scheduled efficiently on a large-scale cluster.

3.2.1 Collaborative Filtering Background

Collaborative filtering techniques are frequently used in recommendation systems. We will use one of their most publicized applications, the Netflix Challenge [34], to provide a quick overview of the two analytical methods we rely upon, Singular Value Decomposition (SVD) and PQ-reconstruction (PQ) [219]. In this case, the goal is to provide valid movie recommendations for Netflix users given the ratings they have provided for various other movies.

The input to the analytical framework is a sparse matrix A , the *utility matrix*, with one row per user and one column per movie. The elements of A are the ratings that users have assigned to movies. Each user has rated only a small subset of movies; this is especially true for new users who may only have a handful of ratings or even none. While there are techniques that address the cold start problem, i.e., providing recommendations to a completely fresh user with no ratings, here we focus on users

for which the system has some minimal input. If we can estimate the values of the missing ratings in the sparse matrix A , we can make movie recommendations: suggest that users watch the movies for which the recommendation system estimates that they will give high ratings with high confidence.

The first step is to apply singular value decomposition (SVD), a matrix factorization method used for dimensionality reduction and similarity identification. Factoring A produces the decomposition to matrices U , V and Σ .

$$A_{m,n} = \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{pmatrix} = U \cdot \Sigma \cdot V^T$$

where

$$U_{m \times r} = \begin{pmatrix} u_{11} & \cdots & u_{1r} \\ u_{21} & \cdots & u_{2r} \\ \vdots & \ddots & \vdots \\ u_{m1} & \cdots & u_{mr} \end{pmatrix}, V_{n \times r} = \begin{pmatrix} v_{11} & v_{12} & \cdots & v_{1r} \\ \vdots & \vdots & \ddots & \vdots \\ v_{n1} & v_{n2} & \cdots & v_{nr} \end{pmatrix}$$

$$\Sigma_{r \times r} = \begin{pmatrix} \sigma_1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \sigma_r \end{pmatrix}$$

are the matrices of left and right singular vectors and the diagonal matrix of singular values.

Dimension r is the rank of matrix A and it represents the number of similarity concepts identified by SVD. For instance, one similarity concept may be that certain movies belong to the drama category, while another may be that most users that liked the movie “Lord of the Rings 1” also liked “Lord of the Rings 2”. Similarity concepts are represented by singular values (σ_i) in matrix Σ and the confidence in a similarity concept by the magnitude of the corresponding singular value. Singular values in Σ are ordered by decreasing magnitude. Matrix U captures the strength of the correlation between a row of A and a similarity concept. In other words, it expresses how users relate to similarity concepts such as the one about liking drama

movies. Matrix V captures the strength of the correlation of a column of A to a similarity concept. In other words, to what extent does a movie fall in the drama category. The complexity of performing SVD on a $m \times n$ matrix is $\min(n^2m, m^2n)$. SVD is robust to missing entries and imposes relaxed sparsity constraints to provide accuracy guarantees. Density less than 1% does not reduce the recommendation accuracy [253].

Before we can make accurate score estimations using SVD, we need the full utility matrix A . To recover the missing entries in A , we use PQ-reconstruction. Building from the decomposition of the initial, sparse A matrix we have $Q_{m \times r} = U$ and $P_{r \times n}^T = \Sigma \cdot V^T$. The product of Q and P^T gives matrix R which is an approximation of A with the missing entries. To improve R , we use Stochastic Gradient Descent (SGD), a scalable and lightweight latent factor model [43, 161, 153, 281] that iteratively recreates A :

$\forall r_{ui}$, where r_{ui} an element of the reconstructed matrix R

$$\epsilon_{ui} = r_{ui} - q_i \cdot p_u^T$$

$$q_i \leftarrow q_i + \eta(\epsilon_{ui}p_u - \lambda q_i)$$

$$p_u \leftarrow p_u + \eta(\epsilon_{ui}q_i - \lambda p_u)$$

until $|\epsilon|_{L_2} = \sqrt{\sum_{u,i} |\epsilon_{ui}|^2}$ becomes marginal.

In the process above η is the learning rate and λ is the regularization factor. The complexity of PQ is linear with the number of r_{ui} and in practice takes up to a few ms for matrices with $m, n \sim 1,000$. Once the dense utility matrix R is recovered we can make movie recommendations. This involves applying SVD to R to identify which of the reconstructed entries reflect strong similarities that enable making accurate recommendations with high confidence.

3.2.2 Classification for Heterogeneity

Overview: We use collaborative filtering to identify how well an incoming application will run on the different hardware platforms available. In this case, the rows in matrix A represent applications, the columns server configurations (SC) and the ratings represent normalized application performance on each server configuration.

As part of an offline step, we select a small number of applications, a few tens, and profile them on all different server configurations. We normalize the performance results and fully populate the corresponding rows of A . This only needs to happen once. If a new configuration is added in the DC, we need to profile these applications on it and add a column in A . In the online mode, when a new application arrives, we profile it for a period of 1 minute on any two server configurations, insert it as a new row in matrix A and use the process described in Section 3.2.1 to derive the missing ratings for the other server configurations.

In this case, Σ represents similarity concepts such as the fact that applications that benefit from SC1 will also benefit from SC3. U captures how an application correlates to the different similarity concepts and V how a server platform correlates to them. Collaborative filtering identifies similarities between new and known applications. Two applications can be similar in one characteristic (they both benefit from high clock frequency) but different in others (only one benefits from a large L3 cache). This is especially common when scaling to large application spaces and several hardware configurations. SVD addresses this issue by uncovering hidden similarities and filtering out the ones less likely to have an impact on the application’s behavior.

The size of the offline training set is important as a certain number of ratings is necessary to satisfy the sparsity constraints of SVD. However, over that number the accuracy quickly levels off and scales well with the number of applications thereafter (smaller fractions for training sets of larger application spaces). For our experiments we use 20 and 50 offline workloads for a 40 and 1,000-server cluster respectively. Additionally, as more incoming applications are added in A the density of the matrix increases and the recommendation accuracy further improves. Note that online training is performed only on two server configurations. This not only reduces the training overhead compared to exhaustive search but since training requires dedicated servers, it also reduces the number of servers necessary for it. In contrast, if we attempted to classify applications through exhaustive profiling, the number of profiling runs would equal the number of server configurations (e.g., 40). For a cloud service with high workload arrival rates, this would be infeasible to support, underlining the importance of keeping training overheads low, something that Paragon does.

Metric	Applications (%)			
	ST	MT	MP	IO
Selected best SC	86%	86%	83%	89%
Selected SC within 5% of best	91%	90%	89%	92%
Correct SC ranking (best to worst)	67%	62%	59%	43%
90% correct SC ranking	78%	71%	63%	58%
50% correct SC ranking	93%	91%	89%	90%
Training & best SC match	28%	24%	18%	22%

Table 3.1: Validation metrics for heterogeneity classification.

Classification is very fast. On a production-class Xeon server, this takes 10-30 msec for thousands of applications and tens of server platforms. We can perform classification for one application at a time or for small groups of incoming applications (*batching*) if the arrival rate is high without impacting accuracy or speed.

Performance scores: We populate A with normalized scores that represent how well an application performs on a server configuration. We use the following performance metrics based on application type:

(a) *Single-threaded workloads:* We use instructions committed per second (IPS) as the initial performance metric. Using execution time would require running applications to completion in the profiling servers, increasing the training overheads. We have verified that using IPS leads to similar classification accuracy as using full execution time. For multi-programmed workloads we use aggregate IPS.

(b) *Multithreaded workloads:* In the presence of spin-locks or other synchronization schemes that introduce active waiting, aggregate IPS can be deceiving [12, 277]. We address this by periodically polling low-overhead performance counters, to detect changes in the register file (read and writes that would denote regular operations other than spinning) and weight-out of the IPS computation such execution segments. We have verified that scheduling with this "useful" IPS leads to similar classification accuracy as using full execution time. When workloads are not known, or multiple workload types are present "useful" IPS is used to drive the scheduling decisions.

The choice of IPS as the base of performance metrics is influenced by our current evaluation which focuses on single-node CPU, memory and I/O intensive programs.

The same methodology holds for higher-level metrics, such as queries per second (QPS), which cover complex multi-tier workloads as well.

Validation: We evaluate the accuracy of heterogeneity classification on a 40-server cluster with 10 server configurations. We use a large set of single-threaded, multi-threaded, multi-programmed and I/O-bound workloads. For details on workloads and server configurations, see Section 3.4. The offline training set includes 20 applications selected randomly from all workload types. The recommendation system achieves 24% performance improvement for single-threaded, 20% for multi-threaded, 38% for multi-programmed, and 40% for I/O workloads on average, while some applications have a 2x performance difference. Table 3.1 summarizes key statistics on the classification quality. Our classifier correctly identifies the best server platform for 84% of workloads and a platform within 5% of optimal for 90%. The predicted ranking of platforms is exactly correct for 58% and almost correct (single reordering) for 65% of workloads. In almost all cases 50% of server configurations are ranked correctly by the classification scheme. Finally, it is important to note that the accuracy does not depend on the two platforms selected for training. The training platform matched the top performing configuration only for 20% of workloads.

We also validate the analytical methods. We compare performance predicted by the recommendation system to performance obtained through experimentation. The deviation is less than 3.8% on average.

3.2.3 Classification for Interference

Overview: There are two types of interference we are interested in: interference that an application can *tolerate* from pre-existing load on a server and interference the application will *cause* on that load. We detect interference due to contention on shared resources and assign a *score* to the sensitivity of an application to a type of interference. To derive sensitivity scores we develop several microbenchmarks, each stressing a specific shared resource with tunable intensity. We run an application concurrently with a microbenchmark and progressively tune up its intensity until the application violates its QoS, which is set at 95% of the performance achieved in

Metric	Percentage (%)
Average sensitivity error across all SoIs	5.3%
Average error for sensitivities < 30%	7.1%
Average error for sensitivities < 60%	5.6%
Average error for sensitivities > 60%	3.4%
Apps with < 5% error	ST: 65% MT: 58%
Apps with < 10% error	ST: 81% MT: 63%
Apps with < 20% error	ST: 90% MT: 89%
SoI with highest error	
for ST: L1 i-cache	15.8%
for MT: LLC capacity	7.8%
Frequency L1 i-cache used as offline SoI	14.6%
Frequency LLC cap used as offline SoI	11.5%
SoI with lowest error	
for ST: network bandwidth	1.8%
for MT: storage bandwidth	0.9%

Table 3.2: Validation metrics for interference classification.

isolation. Applications with high tolerance to interference (e.g., sensitivity score over 60%) are easier to co-schedule than applications with low tolerance (low sensitivity score). Similarly, we detect the sensitivity of a microbenchmark to the interference the application causes by increasing its intensity and recording when the performance of the microbenchmark degrades by 5% compared to its performance in isolation. In this case, high sensitivity scores, e.g., over 60% correspond to applications that cause a lot of interference in the specific shared resource.

Identifying sources of interference (SoI): Co-scheduled applications may contend on a large number of shared resources. We identified ten such sources of interference (SoI) and designed a tunable microbenchmark for each one. SoIs span resources such as memory (bandwidth and capacity), cache hierarchy (L1/L2/L3 and TLBs) and network and storage bandwidth. The same methodology can be expanded to any shared resource.

Collaborative filtering for interference: We classify applications for interference tolerated and caused, using twice the process described in Section 3.2.1. The two utility matrices have applications as rows and SoIs as columns. The elements of the

matrices are the sensitivity scores of an application to the corresponding microbenchmark (sensitivity to tolerated and caused interference respectively). Similarly to classification for heterogeneity, we profile a few applications offline against all SoIs and insert them as dense rows in the utility matrices. In the online mode, each new application is profiled against two randomly chosen microbenchmarks for one minute and its sensitivity scores are added in a new row in each of the matrices. Then, we use SVD and PQ reconstruction to derive the missing entries and the confidence in each similarity concept. This process performs accurate and fast application classification and provides information to the scheduler on which applications should be assigned to the same server (see Section 3.3.2).

Validation: We evaluated the accuracy of interference classification using the single-threaded and multi-threaded workloads and the same systems as for the heterogeneity classification. Table 3.2 summarizes some key statistics on the classification quality. Our classifier, achieves an average error of 5.3% between estimated and measured sensitivity both for tolerated and caused interference across all SoIs. For high values of sensitivity, i.e., applications that tolerate and cause a lot of interference, the error is even lower (3.4%), while for most applications (both single-threaded and multi-threaded) the errors are lower than 5%. The SoIs with the highest errors are the L1 instruction cache for single-threaded workloads and the LLC capacity (L2 or L3) for multi-threaded workloads. The high errors are not a weakness of the classification, since both resources are profiled adequately, but rather of the difficulty to consistently characterize contention in certain shared resources [187]. On the other hand, network and storage bandwidth have the lowest errors, primarily due to the fact that we used CPU and memory intensive workloads for this evaluation.

3.2.4 Putting It All Together

Overall, Paragon requires two short runs (~ 1 minute) on two server configurations to classify incoming applications for heterogeneity. Another two short runs against two microbenchmarks on a high-end server configuration are needed for interference classification. We use a high-end platform to decouple server features from interference

analysis. Running for 1 minute provides some signal on the new workload without introducing significant profiling overheads. In Section 3.3.4 we discuss the issue of workload phases, i.e., transient effects that do not appear in the 1 minute profiling period. Next, we use collaborative filtering to classify the application in terms of heterogeneity and interference, tolerated and caused. This cumulatively requires a few msec even when considering thousands of applications and several tens of platforms or sources of interference. The classification for heterogeneity and interference is performed *in parallel*. For the applications we considered, the overall profiling and classification overheads are 1.2% and 0.09% on average.

Using analytical methods for classification has two benefits; first, we have *strong analytical guarantees* on the quality of the information used for scheduling, instead of relying mainly on empirical observations. The analytical framework provides low and tight error bounds on the accuracy of classification, statistical guarantees on the quality of colocation candidates and detailed characterization of system behavior. Moreover, the scheduler design is workload independent, which means that the analytical or statistical properties the scheme provides hold for any workload. Second, these methods are *computationally efficient, scale well* with the number of applications and server configurations, do not introduce significant training and decision overheads and enable exact complexity evaluation.

3.3 Paragon

3.3.1 Overview

Once an incoming application is classified with respect to heterogeneity and interference, Paragon schedules it on one of the available servers. The scheduler attempts to assign each workload to the server of the best SC and colocate it with applications so that interference is minimized for workloads running on the same server. The scheduler is online and greedy so we cannot make holistic claims about optimality. Nevertheless, the fact that we start with highly accurate classification helps achieve

very efficient schedules. The interference information allows Paragon to pack applications on a subset of servers without significant performance loss¹. The heterogeneity information allows Paragon to assign to each SC only applications that will benefit from its characteristics. Both these properties lead to faster execution, hence resources are freed as soon as possible, making it easier to schedule future applications (more unloaded servers) and perform power management (more idling servers that can be placed in low-power modes).

Figure 3.2 presents an overview of Paragon and its components. The scheduler maintains per-application and per-server state. Per-application state includes information for the heterogeneity and interference classification of every submitted workload. For a DC with 10 SCs and 10 SoIs, we store 64B per application. The per-server state records the IDs of applications running on a server and the cumulative sensitivity to interference (roughly 64B per server). The per-server state needs to be updated as applications are scheduled and, later on, complete. Paragon also needs some storage for the intermediate and final utility matrices and temporary storage for ranking possible candidate servers for an incoming application. Overall, state overheads are marginal and scale logarithmically or linearly with the number of applications (N) and servers (M). In our experiments with thousands of applications and servers, a single server could handle all processing and storage requirements of scheduling².

We present two methods for selecting candidate servers; a fast, greedy algorithm that searches for the optimal candidate, and a statistical scheme of constant runtime that provides strong guarantees on the quality of candidates as a function of examined servers.

3.3.2 Greedy Server Selection

In examining candidates, the scheduler considers two factors: first, which assignments minimize negative interference between the new application and existing load and second, which servers have the best SC for this workload. Decisions are made in this

¹Packing applications with minimal interference should be a property exhibited by any optimal schedule.

²Additional scheduling servers can be used for fault-tolerance.

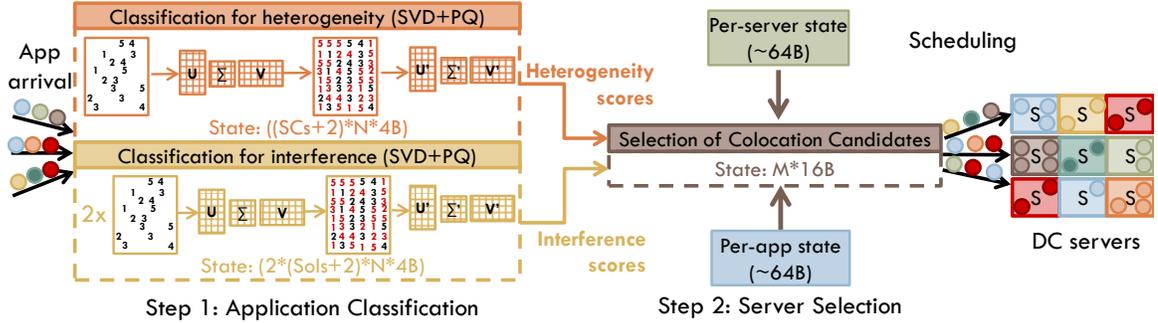


Figure 3.2: The components of Paragon and the state maintained by each component. Overall, the state requirements are marginal and scale linearly or logarithmically with the number of applications (N), servers (M) and configurations.

order; first identifying servers that do not violate QoS and then selecting the best SC between them. This is based on the observation that interference typically leads to higher performance loss than suboptimal SCs.

The greedy scheduler strives to minimize interference, while also increasing server utilization. The scheduler searches for servers whose load can tolerate the interference caused by the new workload and vice versa, the new workload can tolerate the interference caused by the server load. Specifically it evaluates two metrics, $D_1 = t_{server} - c_{newapp}$ and $D_2 = t_{newapp} - c_{server}$, where t is the sensitivity score for tolerated and c for caused interference for a specific SoI. The cumulative sensitivity of a server to caused interference is the sum of sensitivities of individual applications running on it, while the sensitivity to tolerated interference is the minimum of these values. The optimal candidate is a server for which D_1 and D_2 are exactly zero for all SoIs. This implies that there is no negative impact from interference between new and existing applications and that the server resources are perfectly utilized. In practice, a good selection is one for which D_1 and D_2 are bounded by a positive and small ϵ for all SoIs. Large, positive values for D_1 and D_2 indicate suboptimal resource utilization. Negative D_1 and/or D_2 imply violation of QoS and identify poor candidates that should be avoided.

We examine candidate servers for an application in the following way. The process is explained for interference tolerated by the server and caused by the new workload

(D_1) and is exactly the same for D_2 . Given the classification of an application, we start from the resource that is most difficult to satisfy (highest sensitivity score to caused interference). We query the server state and select the server set for which D_1 is non-negative for this SoI. Next, we examine the second SoI in order of decreasing sensitivity scores, filtering out any servers for which D_1 is negative. The process continues until all SoIs have been examined. Then, we take the intersection of candidate server sets for D_1 and D_2 . We now consider heterogeneity. From the set of candidates we select servers that correspond to the best SC for the new workload and from their subset we select the server with $\min(\|D_1 + D_2\|_{L1})$.

As we filter out servers, it is possible that at some point the set of candidate servers becomes empty. This implies that there is no single server for which D_1 and D_2 are non-negative for some SoI. In practice this event is extremely unlikely, but is supported for completeness. We handle this case with backtracking. When no candidates exist the algorithm reverts to the previous SoI and relaxes the QoS constraints until the candidate set becomes non empty, before it continues. If still no candidate is found backtracking is extended to more levels. Given M servers, the worst-case complexity of the algorithm is $O(M \cdot SoI^2)$, since theoretically backtracking might extend all the way to the first SoI. In practice, however, we observe that for a 1000-server system, 89% of applications were scheduled without any backtracking. For 8% of these, backtracking led to negative D_1 or D_2 for a single SoI and for 3% for multiple SoIs. Additionally, we bound the runtime of the greedy search using a timeout mechanism, after which the best server from the ones already examined is selected in the way previously described (best SC and minimum interference deviation). In our experiments timeouts occurred in less than 0.1% of applications and resulted in a server within 10% of optimal.

3.3.3 Statistical Framework for Server Selection

The greedy algorithm selects the best server for an application - or a near-optimal server. However, for very large DCs, e.g., 10-100k servers, the overhead from examining the server state in the first step of the search might become high. Additionally,

the results depend on the active workloads and do not allow strict guarantees on the server quality under any scenario. We now present an alternative, statistical framework for server selection in very large DCs based on sampling, which has constant runtime and enables such guarantees.

Instead of examining the entire server state we sample a small number of servers. We use cryptographic hash functions to introduce randomness in the server selection. We hash the scores of tolerated interference of each server using variations of SHA-1 [153] as different hash functions (h_j) for each SoI to increase entropy. The input to a h_j is a sensitivity score for an SoI and the output a hashed value of that score. Outputs have the same precision as inputs (14bits). This process is done once, unless the load of a server changes. When a new application arrives, we obtain candidate servers by hashing its sensitivity scores to caused interference for each SoI. For example, the input to h_1 for SoI 1 is a . The output will be a new number, b which corresponds to server ID u . Re-hashing b obtains additional IDs of candidate servers. This produces a random subset of the system's servers. After a number of re-hashes the algorithm ranks the examined servers and selects the best one. Candidates are ranked by *colocation quality*, which is a metric of how suitable a given server is for a new workload. For candidate i , colocation quality is defined as:

$$Q_i = [\text{sign}(\sum_{SoIs} (t - c)_i)]|1 - \|t - c\|_1| = [\text{sign}(\sum_{k=1}^{SoIs} (t(k) - c(k))_i)]|1 - \sum_{k=1}^{SoIs} |t(k) - c(k)|_i|$$

t is the original, unhashed sensitivity to tolerated interference for a server and c the original sensitivity to caused interference for the new workload. The *sign* in Q_i reflects whether a server preserves (positive) or violates QoS (negative). The L1 norm of $(t - c)$ reflects how closely the server follows the application's requirements and is normalized to its maximum value, 10, which happens when for all ten SoIs $t = 100\%$ and $c = 0$. High and positive Q_i values reflect better candidates, as the deviation between t and c is small for all SoIs. Poor candidates have small Q_i or even negative when they violate QoS in one or more SoIs. Quality is normalized to the range $[0, 1]$. For example, for unnormalized qualities in the range $[-1.2, 0.8]$ and a candidate with $Q = -1.0$, the normalized quality will be: $\frac{(-1.0 + |min|)}{|max| + |min|} = 0.2/2 = 0.1$.

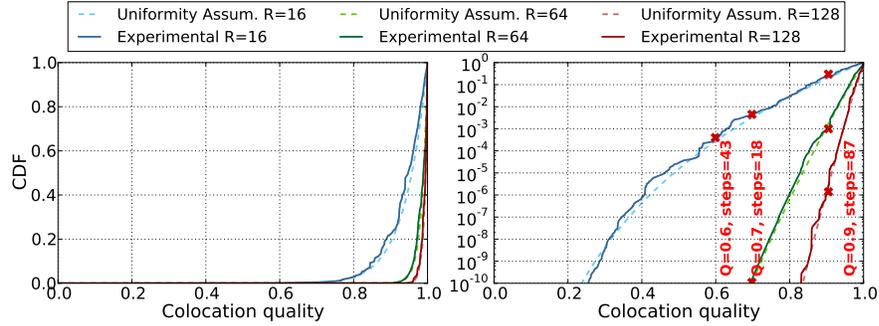


Figure 3.3: Colocation quality distribution ($F(x) = x^R$, where $R = 16, 64$ and 128). Figure 3.3b shows the comparison between the greedy algorithm and the statistical scheme for three colocation candidates of $Q = 0.6, 0.7$ and 0.9 .

We now make an assumption on the distribution of quality values, which we verify in practice. Because of the way candidate servers are selected and the independence between initial workloads, Q_i 's approximate a uniform distribution, for problems with tens of thousands of servers and applications. Figure 3.3a shows the CDF of measured quality for 16, 64 and 128 candidates and the corresponding uniform distributions ($F(x) = x^R$, where R the number of candidates examined) in a system with 1,000 servers. In all cases, the assumption of uniformity holds in practice with small deviations. When we exceed 128 candidates (1/8 of the cluster) the distribution starts deviating from uniform. We have observed that for even larger systems, e.g., a 5,000-server Windows Azure cluster, uniform distributions extend to larger numbers of candidates (up to 512) as well. The probability of a candidate having quality a is $Pr(a) = a^R$. For example, for 128 candidates there is a 10^{-6} probability that no candidate will have quality over 0.9.

We now compare the statistical scheme with the greedy algorithm (Figure 3.3b). While the latter finds a server with quality Q after a random number of steps, the statistical scheme provides strong *guarantees* on the number of candidates required for the same quality. For example, for a candidate with $Q = 0.9$, the greedy algorithm needs 87 steps, but cannot provide ad hoc guarantees on the quality of the result, while the statistical scheme guarantees that for the same requirements, with 64 candidates, there is a 10^{-3} chance that no server has $Q \geq 0.9$. The guarantees become

stricter as the distribution gets skewed towards 1 (more candidates). Therefore, although the statistical scheme cannot guarantee optimality, it shows that examining a small number of servers provides strict guarantees on the obtained quality and makes scheduling efficiency workload independent.

In our 1,000-server experiments, the overhead of executing the greedy algorithm is marginal compared to application execution time (less than 0.1% in most cases), while the statistical scheme induces 0.5-2% overheads due to the computation required for hashing. Because at this scale the greedy algorithm is faster, all results in this work are obtained using greedy search. However, for problems of larger scale the statistical scheme can be more efficient.

3.3.4 Discussion

Workload phases: Application classification in Paragon is performed once for each new workload, using the information from its 1 minute profiling. It is possible that some applications will go through various phases that are not captured during profiling. Hence, the schedule will be suboptimal. We detect such workloads by monitoring their performance scores (e.g., IPS) during execution. If the monitored performance deviates significantly and for long periods of time from the performance predicted by the classification engine, the application may have changed behavior. Upon detection we do one of the following. First, we can avoid scheduling a large number of other workloads on the same server as the interference information for this workload is likely incorrect. Second, if there is a migration mechanism available (process or VM migration), we can clone the workload, repeat the classification from its current execution point and evaluate whether re-scheduling to another server is beneficial. Note that migration can involve significant overheads if the application operates on significant amounts of state. Section 3.5 includes an experiment where workload behavior experiences different phases. We assume that there exists an underlying mechanism, such as vSphere [273], that performs the live migration.

Suboptimal scheduling: A second concern apart from application phases is sub-optimal scheduling, either due to the greedy selection algorithm which assigns applications to servers in a per-workload fashion, or due to pathological behavior in application arrival patterns. Suboptimal scheduling can be detected exactly as the problem of workload phases and can potentially be resolved by re-scheduling several active applications. Although re-scheduling was not needed for the examined applications, Paragon provides a general methodology to detect such deviations and leverage mechanisms like VM migration to re-schedule the sub-optimally scheduled workloads.

Latency-critical applications and workload dependencies: Finally, Paragon does not explicitly consider latency-critical applications or dependencies between application components, e.g., a multi-tier service, such as search or webmail, where tiers communicate and share data. One differentiation in this case comes from the metrics the scheduler must consider. It is possible that the interference classification should use microbenchmarks that aim to degrade the per-query latency as opposed to the workload’s throughput. Another differentiation comes from the possible workload scenarios. One scenario can involve a latency-critical application running as the primary process, e.g., memcached, and the remaining server capacity being allocated to best-effort applications, such as analytics or background processes using Paragon. A different scenario is one where a throughput-bound distributed workload, e.g., MapReduce runs with high priority and the remaining server capacity is used by instances of a latency-critical application. Paragon does not currently enforce fine-grain priorities between application components or user requests, or optimize for shared data placement, which might be beneficial for these scenarios.

3.4 Methodology

Server systems: We evaluated Paragon on a small local cluster and three major cloud computing services. Our local cluster includes servers of ten different configurations shown in Table 6.1. We also show how many servers of each type we use. Note that these configurations range from high-end Xeon systems to low-power Atom-based boards. There is a wide range of core counts, clock frequencies and memory

Server Type	GHz	sockets	cores	L1(KB)	LLC(MB)	mem(GB)	#
Xeon L5609	1.87	2	8	32/32	12	24 DDR3	1
Xeon X5650	2.67	2	12	32/32	12	24 DDR3	2
Xeon X5670	2.93	2	12	32/32	12	48 DDR3	2
Xeon L5640	2.27	2	12	32/32	12	48 DDR3	1
Xeon MP	3.16	4	4	16/16	1	8 DDR2	5
Xeon E5345	2.33	1	4	32/32	8	32 FB-DIMM	8
Xeon E5335	2.00	1	4	32/32	8	16 FB-DIMM	8
Opteron 240	1.80	2	2	64/64	2	4 DDR2	7
Atom 330	1.60	1	2	32/24	1	4 DDR2	5
Atom D510	1.66	1	2	32/24	1	8 DDR2	1

Table 3.3: Main characteristics of the servers of the local cluster. The total core count is 178 for 40 servers of 10 different server configurations.

capacities and speeds present in the cluster.

For the cloud-based clusters we used exclusive (reserved) server instances, i.e., no other users had access to these servers. We verified that no external scheduling decisions or actions such as auto-scaling or workload migration are performed during the course of the experiments. We used 1,000 servers on Amazon EC2 [13] with 14 different server configurations, ranging from small, low-power, dual-core machines to high-end, quad-socket, multi-core servers with hundreds of GBs of memory. All 1,000 machines are private, i.e., there is no interference in the experiments from external workloads. We also conducted experiments with 500 servers on Windows Azure [279] with 8 different server configurations and 100 servers on Google Compute Engine [110] with 4 server configurations.

Schedulers: We compared Paragon to three alternative schedulers. First, we evaluate a baseline scheduler that preserves an application’s core and memory requirements but ignores both its heterogeneity and interference profiles. In this case, applications are assigned to the *least-loaded* (LL) machine. Second, we examine a *heterogeneity-oblivious* (NH) scheme that uses the interference classification in Paragon to assign applications to servers without visibility in their server platforms. Finally, we evaluate an *interference-oblivious* (NI) scheme that uses the heterogeneity classification in Paragon but has no insight on workload interference. The overheads for the heterogeneity and interference-oblivious schemes are the corresponding classification and

server selection overheads.

Workloads: We used 29 single-threaded (ST), 22 multi-threaded (MT) and 350 multi-programmed (MP) workloads and 25 I/O-bound workloads. We use the full SPEC CPU2006 suite and workloads from PARSEC [40] (*blackscholes, bodytrack, facesim, ferret, fluidanimate, raytrace, swaptions, canneal*), SPLASH-2 [282] (*barnes, fft, lu, ocean, radix, water*), BioParallel [150] (*genenet, svm*), Minebench [198] (*semphy, plsa, kmeans*) and SPECjbb (*2, 4 and 8-warehouse instances*). For multiprogrammed workloads, we use 350 mixes of 4 applications, based on the methodology in [228]. The I/O-bound workloads are data mining applications, such as clustering and recommender systems [219], in Hadoop and Matlab running on a single-node. Workload durations range from minutes to hours. For workload scenarios with more than 426 applications we replicated these workloads with equal likelihood (1/4 ST, 1/4 MT, 1/4 MP, 1/4 I/O) and randomized their interleaving.

Workload scenarios: To explore a wide range of behaviors, we used the applications listed above to create multiple workload scenarios. Scenarios vary in the number, type and inter-arrival times of submitted applications. The load is classified based on its relation to available resources; low: the required core count is significantly lower than the available processor resources; high: the required core count approaches the load the system can support but does not surpass it; and oversubscribed: the required core count often exceeds the system’s capabilities, i.e., certain machines are oversubscribed.

For the *small-scale* experiments on the local cluster we examine four workload scenarios. First, a *low load* scenario with 178 applications, selected randomly from the pool of workloads, which are submitted with 10 sec inter-arrival times. Second, a *medium load* scenario with 178 applications, randomly selected as before and submitted with inter-arrival times that follow a Gaussian distribution with $\mu = 10$ sec and $\sigma^2 = 1.0$. Third, a *high load* scenario with 178 workloads, each corresponding to a sequence of three applications with varying memory loads. Each application goes through three phases; first medium, then high and again medium memory load. Workloads are submitted with 10 sec intervals. Finally, we examine a scenario, where

178 randomly-chosen applications arrive with 1 sec intervals. Note that the last scenario is an *over-subscribed* one. After a few seconds, there are not enough resources in the system to execute all applications concurrently, and subsequent submitted applications are queued.

For the *large-scale* experiments on EC2 we examine three workload scenarios; a *low load* scenario with 2,500 randomly-chosen applications submitted with 1 sec intervals, a *high load* scenario with 5,000 applications submitted with 1 sec intervals and an *oversubscribed scenario* where 7,500 workloads are submitted with 1 sec intervals and an additional 1,000 applications arrive in burst (less than 0.1 sec intervals) after the first 3,750 workloads.

3.5 Evaluation

3.5.1 Comparison of Schedulers: Small Scale

QoS guarantees: Figure 3.4 summarizes the performance results across the 178 workloads on the 40-server cluster for the *medium load* scenario where application arrivals follow a Gaussian distribution. Applications are ordered in the x-axis from worst to best-performing workload. The y-axis shows the performance (execution time) normalized to the performance of an application when it is running in the best platform in isolation (without interference). Each line corresponds to the performance achieved with a different scheduler. Overall, Paragon (P) outperforms the other schedulers, in terms of preserving QoS (95% of optimal performance), and bounding performance degradation when QoS requirements cannot be met. 78% of workloads maintain their QoS with Paragon, while the heterogeneity-oblivious (NH), interference-oblivious (NI) and least-loaded (LL) schedulers provide similar guarantees only for 23%, 19% and 7% of applications respectively. Even more, for the case of the least-loaded scheduler some applications failed to complete due to memory exhaustion on the server. Similarly, while the performance degradation with Paragon is smooth (94% of workloads have less than 10% degradation), the other three schedulers dramatically degrade performance for most applications, in almost

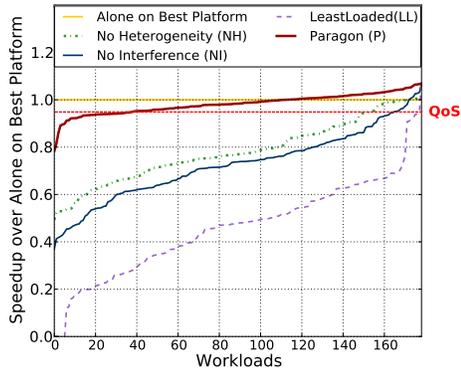


Figure 3.4: Performance impact from scheduling with Paragon for *medium load*, compared to heterogeneity and/or interference-oblivious schedulers. Application arrival times follow a Gaussian distribution. Applications are ordered from worst to best.

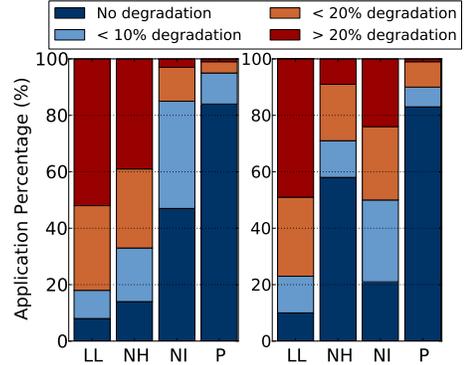


Figure 3.5: Breakdown of decision quality for heterogeneity (left) and interference (right) for the medium load on the local cluster. Applications are divided based on performance degradation induced by the decisions made by each of the schedulers.

linear fashion with the number of workloads. For this scenario, the heterogeneity and interference-oblivious schedulers perform almost identically, although ignoring interference degrades performance slightly more. This is due to workloads that arrive at the peak of the Gaussian distribution, when the cluster’s resources are heavily utilized. For the same workloads, Paragon limits performance degradation to less than 10% in most cases. This figure also shows that a small number of workloads experience speedups compared to their execution in isolation. This is a result of cache effects or instruction prefetching between similar co-scheduled workloads. We expect positive interference to be less prevalent for a more diverse application space.

Scheduling decision quality: Figure 3.5 explains why Paragon achieves better performance. Each bar represents a percentage of applications based on the performance degradation they experience due to the quality of decisions of each of the four schedulers in terms of platform selection (left) and impact from interference. Blue bars reflect good and red bars poor scheduling decisions. In terms of platform decisions, the least-loaded scheduler (LL) maps applications to servers with no heterogeneity considerations, thus it significantly degrades performance for most applications. The

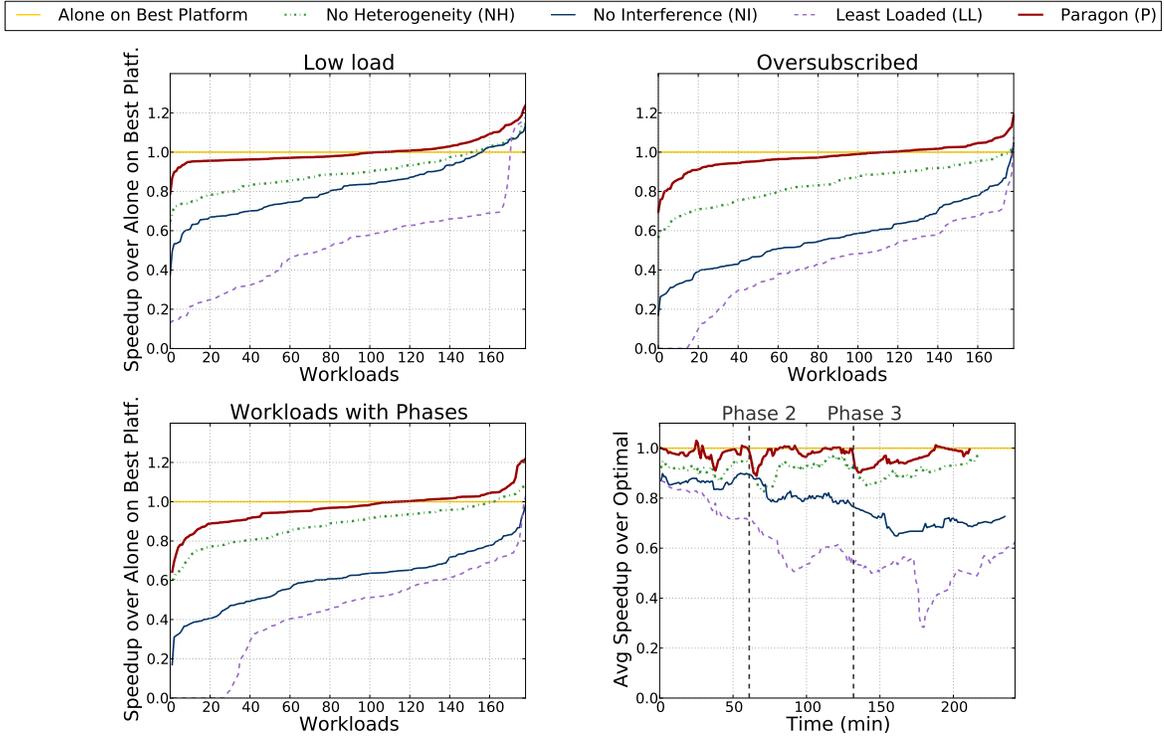


Figure 3.6: Performance comparison between the four schedulers for three workload scenarios: low, oversubscribed and workloads with phases (Figure 3.6(a, b, c)) and performance over time for the scenario where workloads experience phases (Figure 3.6d).

heterogeneity-oblivious (NH) scheduler assigns more than 40% of workloads to sub-optimal server platforms, although fewer than LL, as it often steers workloads to high-end server platforms that tend to tolerate more interference. However, as these servers become saturated, applications that would benefit from them are scheduled suboptimally and NH ends up making poor quality assignments afterwards. On the other hand, the schedulers that account for heterogeneity explicitly (interference-oblivious (NI) and Paragon (P)) have much better decision quality. NI induces no degradation to 47% of workloads and less than 10% for an additional 38%. The reason why NI does not behave better in terms of platform selection is that it has no input on interference, therefore it assigns most workloads to the best server configurations. As these machines become saturated, destructive interference increases and

performance degrades, although, unlike NH, which selects a random server configuration next, NI selects the server configurations that is ranked second for a workload. Finally, Paragon outperforms the other schedulers and assigns 84% of applications to their optimal server configuration.

The right part in Figure 3.5 shows decision quality with respect to interference. LL behaves the worst for similar reasons, while NI is slightly better than LL since it assigns more applications to high-end server configuration, that are more likely to tolerate interference. NH outperforms NI as expected, since NI ignores interference altogether. Paragon assigns 83% of applications to servers that induce no negative interference. Considering both graphs establishes why Paragon significantly outperforms the other schedulers, as it has better decision quality both in terms of heterogeneity and interference.

Other workload scenarios: Figure 3.6 compares Paragon to the three schedulers for the other three scenarios; low load, oversubscribed, and workloads with phases. For low load, performance degradation is small for all schedulers, although LL degrades performance by 46% on average. Since the cluster can easily accommodate the load of most workloads, classifying incoming applications has a smaller performance impact. Nevertheless, Paragon outperforms the other three schedulers and achieves 99% of optimal performance on average. It also improves resource efficiency during low load by completing the scenario 15% faster than the least-loaded scheduler. For the oversubscribed scenario, Paragon guarantees QoS for the largest workload fraction, 75% and bounds degradation to less than 10% for 99% of workloads. In this case, accounting for interference is much more critical than accounting for heterogeneity as the system’s resources are fully utilized.

Finally, for the case where workloads experience phases, we want to validate two expectations. First, Paragon should outperform the other schedulers, since it accounts for heterogeneity and interference (66% of workloads preserve their QoS). Second, Paragon should adapt to the changes in workload behavior, by detecting deviations from the expected IPS, re-classifying the offending workloads and re-scheduling them if a more suitable server is available. To verify this, in Figure 3.6d we show the average performance for each scheduler over time. The points where workloads start

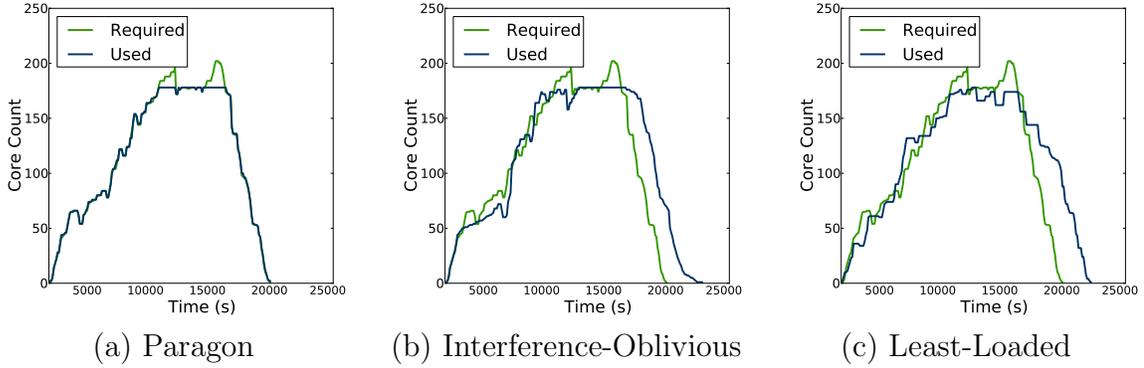


Figure 3.7: Comparison of resource activity between Paragon, the interference-oblivious and the least-loaded scheduler. Plots show the required and allocated core count at each moment.

changing phases are denoted with vertical lines. First, at phase change, Paragon induces much less degradation than the other schedulers, because applications are assigned to appropriate servers to begin with. Second, Paragon recovers much faster and better from the phase change. Performance rebounds to values close to 1 as the deviating workloads are re-scheduled to appropriate servers, while the other schedulers achieve progressively worse average performance.

Resource allocation: Ideally, the scheduler should closely follow application resource requirements (cores, cache capacity, memory bandwidth, etc.) and provide them with the minimum number of servers. This improves performance (applications execute as fast as possible without interference) and reduces overprovisioning (number of servers used, periods for which they are active). The latter particularly benefits the DC operator, as it reduces both capital and operational expenses. A smaller number of servers needs to be purchased to support a certain load (capital savings). During low load, many servers can be turned off to save energy (operational savings).

Figure 3.7a shows how Paragon follows the resource requirements for the medium load scenario shown in Figure 3.4. The green line shows the ideally required core count of active applications based on arrival rate and ideal execution time and the blue line the allocated core count by Paragon. Because the scheduler tracks application behavior in terms of heterogeneity and interference it is able to follow their requirements with minimal deviation (less than 3.5%), excluding periods when the

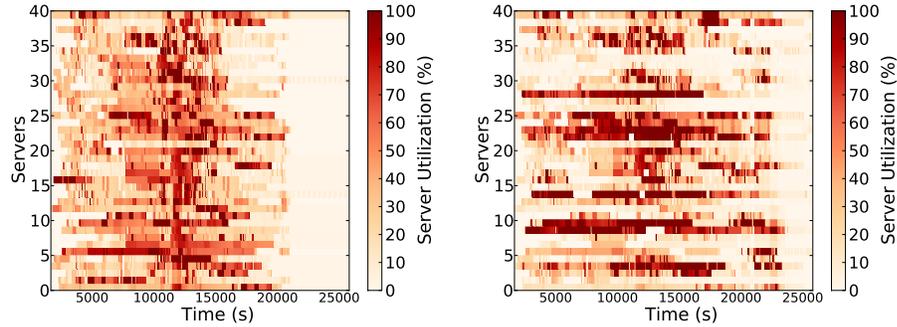


Figure 3.8: Comparison of server utilization for Paragon (left) and the interference-oblivious scheduler (right). Darker colors correspond to higher CPU utilization.

system is oversubscribed and the required cores exceed the total number of cores in the system. In comparison, NI (Figure 3.7b) and similarly for NH, either overprovisions or oversubscribes servers, resulting in increased execution time; per-application and for the overall scenario. Finally, Figure 3.7c shows the resource allocation for the least-loaded scheduler. There is significant deviation, since the scheduler ignores both heterogeneity and interference. All cores are used but in a suboptimal manner. Hence, execution times are increased for individual workloads and the overall scenario. Total execution time increases by 15%, but more importantly per-application time degrades (Figure 3.4), which is harmful both for users and DC operators.

Server utilization: In Figure B.9 we plot heat maps of the server utilization over time for Paragon and the interference-oblivious (NI) scheduler. Server utilization is defined as average CPU utilization across the cores of a server. For Paragon, utilization is high in the middle of the scenario when many applications are active (47% higher than without colocation), and returns to zero when the scenario finishes. In this case, resource usage improves compared to the interference-oblivious scheduler without performance degradation due to interference. On the other hand, NI keeps server utilization high in some servers and underutilizes others, while violating per-application QoS and extending the scenario’s execution time. This is undesirable both for the user who gets lower performance and for the DC operator, since the high utilization in certain servers does not translate to faster execution time, adhering scalability to servicing more workloads.

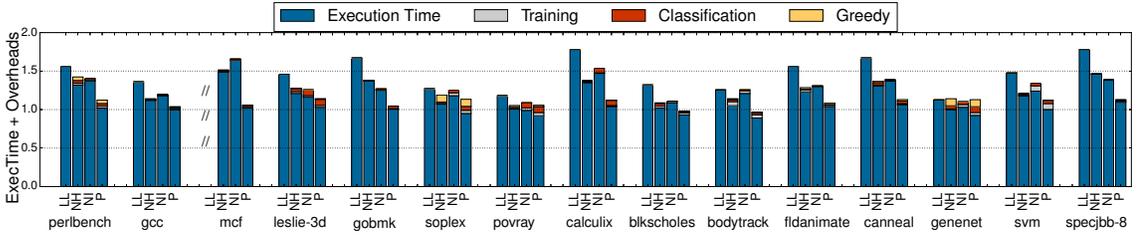


Figure 3.9: Execution time breakdown for selected single-threaded and multi-threaded applications in the medium load scenario.

Scheduling overheads: Finally, we evaluate the total scheduling overheads for the various schemes. These include the overheads of offline training, classification and server selection using the greedy algorithm. Figure 3.9 shows the execution time breakdown for selected single-threaded and multi-threaded applications. These applications are representative of workloads submitted throughout the execution of the medium load scenario. All bars are normalized to the execution time of the application in isolation in the best server configuration. Training and classification for heterogeneity and interference are performed in parallel so there is a single bar for each, for every workload. There is no bar for the least-loaded scheduler for *mcf*, since it was one of the benchmarks that did not terminate successfully. Paragon achieves lower execution times for the majority of applications and close to optimal. The overheads of the recommendation system are low; 1.2% for training and 0.09% for classification. The overheads of the greedy algorithm are less than 0.1% in most cases with the exceptions of *soplex* and *genenet* that required extensive backtracking which was handled with a timeout. Overall, Paragon performs accurate classification and efficient scheduling within 1 minute of the application’s arrival, which is marginal for most workloads.

3.5.2 Comparison of Schedulers: Large Scale

Performance impact: Figure 3.10 shows the performance for the three workload scenarios on the 1,000-server EC2 cluster. Similar to the results on the local cluster, the *low load* scenario, in general, does not create significant performance challenges.

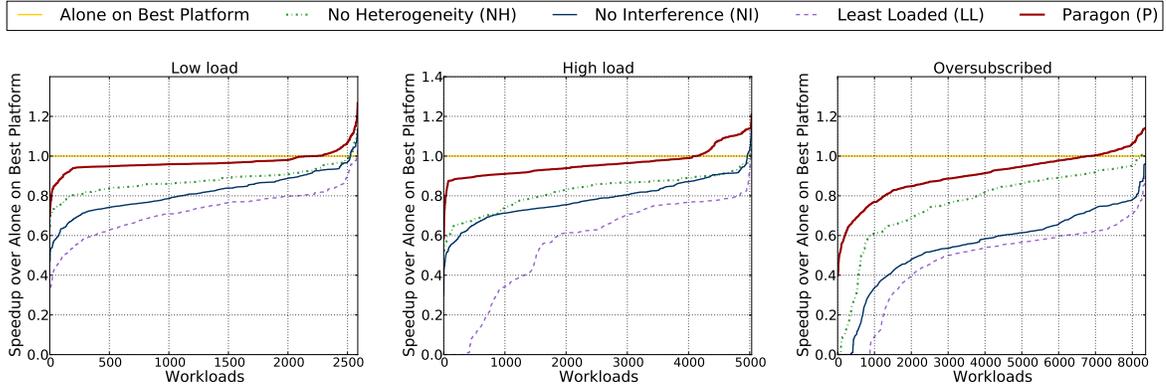


Figure 3.10: Performance comparison between the four schedulers, for three workload scenarios on 1,000 EC2 servers.

Nevertheless, Paragon outperforms the other three schemes, it maintains QoS for 91% of workloads and achieves on average 0.96 of the performance of a workload running in isolation in the best server configuration. When moving to the case of *high load*, the difference between schedulers becomes more obvious. While the heterogeneity and interference-oblivious schemes degrade performance by an average of 22% and 34% and violate QoS for 96% and 97% of workload respectively, Paragon degrades performance only by 4% and *guarantees QoS* for 61% of workloads. The least-loaded scheduler degrades performance by 48% on average, while some applications do not terminate (crash). The differences in performance are larger for workloads submitted when the system is heavily loaded and becomes oversubscribed. Although, we simply queue applications in FIFO order until resources become available, Paragon bounds performance degradation (only 0.6% of workloads degrade more than 20%), since it co-schedules workloads that minimize destructive interference. We plan to incorporate a better admission control protocol in the scheduler in future work.

Finally, for the oversubscribed case, NH, NI and LL dramatically degrade performance for most workloads, while the number of applications that do not terminate successfully increases to 10.4%. Paragon, on the other hand, provides strict QoS guarantees for 52% of workloads, while the other schedulers provide similar guarantees only for 5%, 1% and 0.09% of workloads respectively. Additionally, Paragon

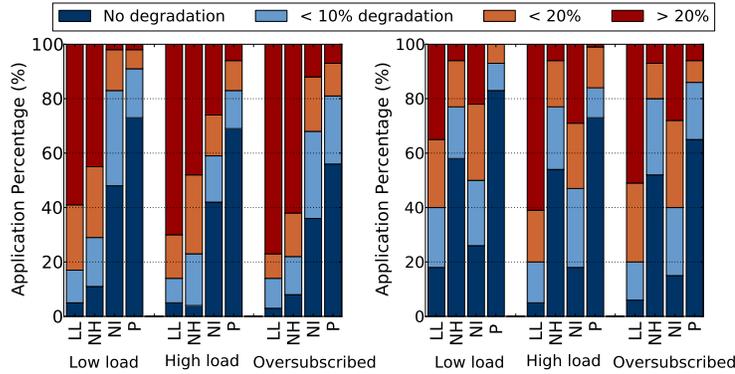


Figure 3.11: Breakdown of decision quality in terms of heterogeneity (left) and interference for the three EC2 scenarios.

limits degradation to less than 10% for an additional 33% of applications and maintains performance degradation moderate (no cliffs in performance such as for NH in applications [1-1000]).

Decision quality: Figure 3.11 shows a breakdown of the decision quality of the different schedulers for heterogeneity (left) and interference (right) across the three experiments. LL induces more than 20% performance degradation to most applications, both in terms of heterogeneity and interference. NH has low decision quality in terms of platform selection, while NI causes performance degradation by collocating unsuitable applications. The errors increase as we move to scenarios of higher load. Paragon decides optimally for 65% of applications for heterogeneity and 75% for interference on average, significantly higher than the other schedulers. It also constrains decisions that lead to larger than 20% degradation due to interference to less than 8% of workloads. The results are consistent with the findings for the small-scale experiments.

Resource allocation: Figure 3.12 shows why this deviation exists. From left to right we show the graphs for low, high, and oversubscribed load. The yellow line represents the required core count based on the applications running at a snapshot of the system, while the other four lines show the allocated core count by each of the schedulers. Since Paragon optimizes for increased utilization within QoS constraints, it follows the application requirements closely. It only deviates when the required

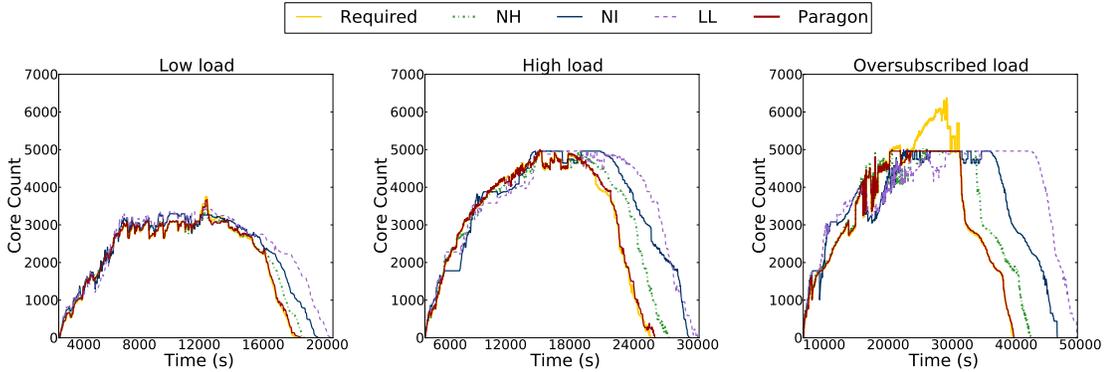


Figure 3.12: Comparison of required and performed core allocation between Paragon and the other three schedulers for the three workload scenarios on EC2. The total number of cores in the system is 4960.

core count exceeds the resources available in the system. NH has mediocre accuracy, while NI and LL either significantly overprovision the number of allocated cores, or oversubscribe certain servers. There are two important points in these graphs: first, as the load increases the difference in execution time exceeds the optimal one, which Paragon approximates with minimal deviation. Second, for higher loads, the errors in core allocation increase dramatically for the other three schedulers, while for Paragon the average deviation remains constant, excluding the part where the system is oversubscribed.

Windows Azure & Google Compute Engine: We validate our results on a 500-server Azure and a 100-server Compute Engine (GCE) cluster (Figure 3.13). We run a scenario with 2,500 and 500 workloads respectively. In Azure, Paragon achieves 94.3% of the performance in isolation and maintains QoS for 61% of workloads, while the other three schedulers provide the same guarantees for 1%, 2% and 0.7% of workloads. Additionally, this was the only time where NI outperformed NH, most likely due to the wide variation between server configurations which increases the importance of accounting for heterogeneity. In the GCE cluster, which has only 4 server configurations, workloads exhibit mediocre benefits from heterogeneity-aware scheduling (7% over random), while the majority of gains comes from accounting for interference. Overall, Paragon achieves 96.8% of optimal performance and NH

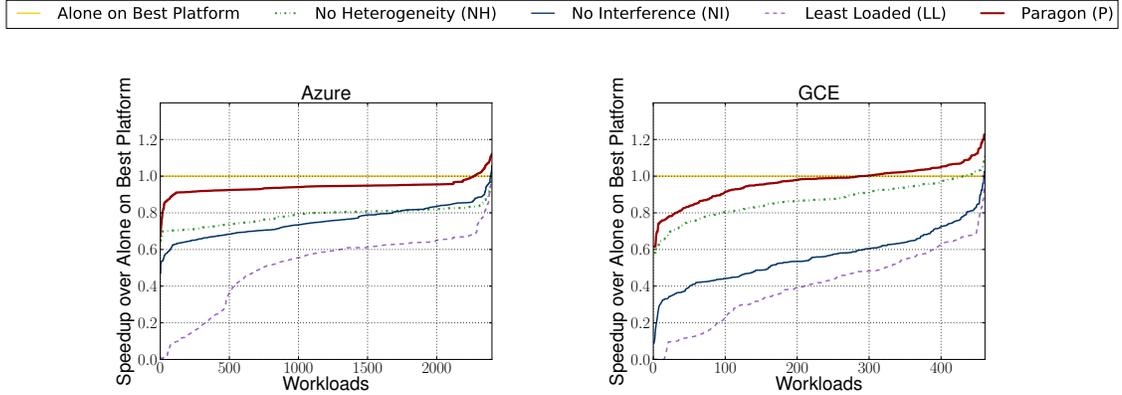


Figure 3.13: Performance comparison between the schedulers on the Windows Azure and Google Compute Engine (GCE) clusters.

90%. The consistency between experiments, despite the different cluster configurations and underlying hardware, shows the robustness of the analytical methods that drive Paragon.

3.6 Related Work

We discuss work relevant to Paragon in the areas of DC scheduling, VM management and workload rightsizing. We also present related work from scheduling for heterogeneous multi-core chips.

Datacenter scheduling: Recent work on DC scheduling has highlighted the importance of platform heterogeneity and workload interference. Mars et al. [188, 187] showed that the performance of Google workloads can vary by up to 40% due to heterogeneity even when considering only two SCs and up to 2x due to interference even when considering only two colocated applications. In [188], they present an offline scheme that used combinatorial optimization to select the proper SC for each workload. In [187], they present an offline, two-step method to characterize the sensitivity of workloads to memory pressure and the stress each application exercises to the memory subsystem. Govindan et al. [120] also present a scheme to quantify the effects of cache interference between consolidated workloads, although they require

access to physical memory addresses. Finally, Nathuji et al. [200] present a control-based resource allocation scheme that mitigates the effects of cache, memory and hardware prefetching interference of co-scheduled workloads. In Paragon, we extend the concepts of heterogeneity and interference-aware DC scheduling in several ways. We provide an online, highly-accurate and low-overhead methodology that classifies applications for both heterogeneity and interference across multiple resources. We also show that our classification engine allows for efficient, online scheduling without using computationally intensive techniques which require exhaustive search between colocation candidates.

VM management: VM management systems such as vSphere [273], XenServer [4] or the VM platforms on EC2 [13] and Windows Azure [279] can schedule diverse workloads submitted by a large number of users on the available servers. In general, these platforms account for application resource requirements which they learn over time by monitoring workload execution. Paragon can complement such systems by making efficient scheduling decisions based on heterogeneity and interference and detecting when an application should be considered for migration (re-scheduling).

Resource management and rightsizing: There has been significant work on resource allocation in virtualized and non-virtualized large-scale DCs, including Mesos [139], Rightscale [224], resource containers [26], Dejavu [266] and the work by Chase et al. [54]. Mesos performs resource allocation between distributed computing frameworks like Hadoop or Spark [139]. Rightscale automatically scales out 3-tier applications to react to changes in the load in Amazon’s cloud service [24]. Dejavu serves a similar goal by identifying a few workload classes and based on them, reuses previous resource allocations to minimize reallocation overheads [266]. Zhu et al. [300] present a resource management scheme for virtualized DCs that preserves SLAs and Gmach et al. [117] a resource allocation scheme for DC applications that relies on the ability to predict their behavior a priori. In general, Paragon is complementary to resource allocation and rightsizing systems. Once such a system determines the amount of resources needed by an application (e.g., number of servers, memory capacity, etc.), Paragon can classify and schedule it on the proper hardware platform in a way that minimizes interference. Currently, Paragon focuses on online scheduling of previously

unknown workloads. We will consider how to integrate Paragon with a rightsizing system for scheduling long running, 3-tier services in future work.

Scheduling for heterogeneous multi-core chips: Finally, scheduling in heterogeneous CMPs shares some concepts and challenges with scheduling in heterogeneous DCs, therefore some of the ideas in Paragon can be applied in heterogeneous CMP scheduling as well. Fedorova et al. [102, 101] discuss OS level scheduling for heterogeneous multi-cores as having the following three objectives: optimal performance, core assignment balance and response time fairness. Shelepov et al. [239] present a scheduler that exhibits some of these features and is simple and scalable, while Craeynest et al. [263] use performance statistics to estimate which workload-to-core mapping is likely to provide the best performance. DC scheduling also has similar requirements as applications should observe their QoS, resource allocation should follow application requirements closely and fairness between co-scheduled workloads should be preserved. Given the increasing number of cores per chip and co-scheduled tasks, techniques such as those used for the classification engine of Paragon can be applicable when deciding how to schedule applications to heterogeneous cores as well.

3.7 Conclusions

In this chapter, we have presented Paragon, a scalable scheduler for DCs that is both heterogeneity and interference-aware. Paragon is derived from validated analytical methods, such as collaborative filtering to quickly and accurately classify incoming applications with respect to platform heterogeneity and workload interference. Classification uses minimal information about the new application and relies mostly on information from previously scheduled workloads. The output of classification is used by a greedy scheduler to assign workloads to servers in a manner that maximizes application performance and optimizes resource usage. We have evaluated Paragon with both small and large-scale systems. Even for very demanding scenarios, where heterogeneity and interference-agnostic schedulers degrade performance for up to 99.9% of workloads, Paragon maintains QoS guarantees for 52% of the applications and bounds degradation to less than 10% for an additional 33% out of 8500 applications

on a 1,000-server cluster. Paragon preserves QoS guarantees while improving server utilization, hence it benefits both the DC operator, who achieves perfect resource use and the user, who gets the best performance. In the following chapter we discuss how a similar approach can be applied towards resource provisioning in large-scale datacenters.

Chapter 4

Quasar: QoS-Aware and Resource Efficient Cluster Management

4.1 Introduction

In the previous chapter, we discussed how fast data mining techniques can be used to manage platform heterogeneity and workload interference to improve performance in datacenter workloads. Nevertheless, managing the type of resources assigned to a new application is not sufficient to resolve the underutilization problem in datacenters. In fact, most cloud facilities operate at very low utilization, even when using cluster management frameworks that enable cluster sharing across workloads [29, 223]. In this chapter we discuss the reasons behind underutilization, and propose a new cluster management approach that significantly improves resource efficiency. In Figure 4.1, we present a utilization analysis for a production cluster at Twitter with thousands of servers, managed by Mesos [139] over one month. The cluster mostly hosts user-facing services. The aggregate CPU utilization is consistently below 20%, even though reservations reach up to 80% of total capacity (Figure 4.1.a). Even when looking at individual servers, their majority does not exceed 50% utilization on any week (Figure 4.1.c). Typical memory use is higher (40-50%) but still differs from the reserved capacity. Figure 4.1.d shows that very few workloads reserve the right amount of resources (compute resources shown here, similar for memory); most

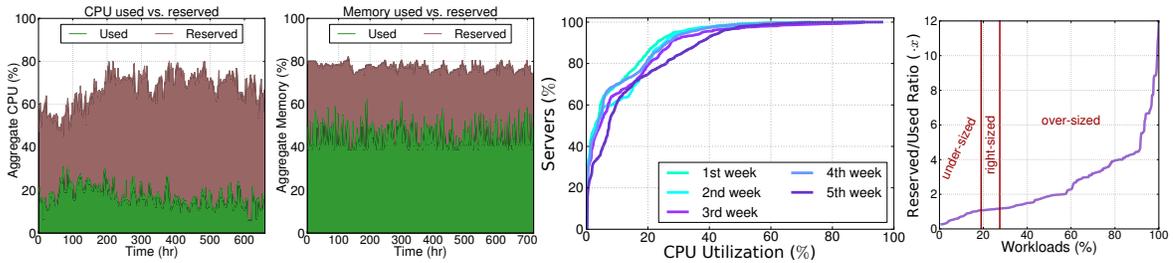


Figure 4.1: Resource utilization over 30 days for a large production cluster at Twitter managed with Mesos. (a) and (b): utilization vs reservation for the aggregate CPU and memory capacity of the cluster; (c) CDF of CPU utilization for individual servers for each week in the 30 day period; (d) ratio of reserved vs used CPU resources for each of the thousands of workloads that ran on the cluster during this period.

workloads (70%) overestimate reservations by up to 10x, while many (20%) underestimate reservations by up to 5x. Similarly, Reiss et al. showed that a 12,000-server Google cluster managed with the more mature Borg system consistently achieves aggregate CPU utilization of 25-35% and aggregate memory utilization of 40% [223]. In contrast, reserved resources exceed 75% and 60% of available capacity for CPU and memory respectively.

Twitter and Google are in the high end of the utilization spectrum. Utilization estimates are even lower for cloud facilities that do not colocate workloads the way Google and Twitter do with Borg and Mesos respectively. Various analyses estimate industry-wide utilization between 6% [63] and 12% [265, 109]. A recent study estimated server utilization on Amazon EC2 in the 3% to 17% range [177]. Overall, low utilization is a major challenge for cloud facilities. Underutilized servers contribute to capital expenses and, since they are not energy proportional [169, 191], to operational expenses as well. Even if a company can afford the cost, low utilization is still a scaling limitation. With many cloud DCs consuming 10s of megawatts, it is difficult to add more servers without running into the limits of what the nearby electricity facility can deliver.

We focus on increasing resource utilization in datacenters through better cluster management. The manager is responsible for providing resources to various workloads in a manner that achieves their performance goals, while maximizing the utilization of available resources. The manager must make two major decisions; first allocate the

right amount of resources for each workload (*resource allocation*) and then select the specific servers that will satisfy a given allocation (*resource assignment*). While there has been significant progress in cluster management frameworks [98, 139, 232, 270], there are still major challenges that limit their effectiveness in concurrently meeting application performance and resource utilization goals. First, it is particularly difficult to determine the resources needed for each workload. The load of user-facing services varies widely within a day, while the load of analytics tasks depends on their complexity and their dataset size. Most existing cluster managers side-step allocation altogether, requiring users or workloads to express their requirements in the form of a reservation. Nevertheless, the workload developer does not necessarily understand the physical resource requirements of complex codebases or the variations in load and dataset size. As shown in Figure 4.1.d, only a small fraction of the workloads submitted to the Twitter cluster provided a right-sized reservation. Undersized reservations lead to poor application performance, while oversized reservations lead to low resource utilization.

Equally important, resource allocation and resource assignment are fundamentally linked. The first reason is heterogeneity of resources, which is quite high as servers get installed and replaced over the typical 15-year lifetime of a DC [29, 76]. A workload may be able to achieve its current performance goals with ten high-end or twenty low-end servers. Similarly, a workload may be able to use low-end CPUs if the memory allocation is high or vice versa. The second reason is interference between colocated workloads that can lead to severe performance losses [188, 285]. This is particularly problematic for user-facing services that must meet strict, tail-latency requirements (e.g., low 99th percentile latency) under a wide range of traffic scenarios ranging from low load to unexpected spikes [68]. Naïvely collocating these services with low-priority, batch tasks that consume any idling resources can lead to unacceptable latencies, even at low load [188]. This is the reason why cloud operators deploy low-latency services on dedicated servers that operate at low utilization most of the time. In facilities that share resources between workloads, users often exaggerate resource reservations to side-step performance unpredictability due to interference. Finally, most cloud facilities are large and involve thousands of servers and workloads, putting tight

constraints on the complexity and time that can be spent making decisions [232]. As new, unknown workloads are submitted, old workloads get updated, new datasets arise, and new server configurations are installed, it is impractical for the cluster manager to analyze all possible combinations of resource allocations and assignments.

We present *Quasar*, a cluster manager that maximizes resource utilization while meeting performance and QoS constraints for each workload. Quasar includes three key features. First, it shifts from a *reservation-centric* to a *performance-centric* approach for cluster management. Instead of users expressing low-level resource requests to the manager, Quasar allows users to communicate the performance constraints of the application through a high-level, declarative interface. Performance constraints are expressed in terms of throughput and/or latency, depending on the application type. This high-level interface allows Quasar to determine the *least amount of the available resources* needed to meet performance constraints at any point, given the current state of the cluster in terms of available servers and active workloads. The allocation varies over time to adjust to changes in the workload or system state. The performance-centric approach simplifies both the user and cloud manager's roles as it removes the need for exaggerated reservations, allows transparent handling of unknown, evolving, or irregular workloads, and provides additional flexibility towards cost-efficient allocation.

Second, Quasar uses *fast classification techniques to determine the impact of different resource allocations and assignments on workload performance*. This problem is much more complex than the one addressed in Chapter 3, since apart from heterogeneity and interference, the system must also determine the amount of resources an application should receive within a node, the ratio of resources in the allocation, the number and topology of nodes in the case of distributed applications, and the way parameters in frameworks such as Hadoop and Spark should be configured. Exhaustively exploring the space would require billions of profiling runs for clusters with a few hundred nodes. Instead in Quasar, by combining a small amount of profiling information from the workload itself with the large amount of data from previously-scheduled workloads, we can quickly and accurately generate the information needed for efficient resource assignment and allocation without the need for a priori analysis

of the application and its dataset. Applying classification to cluster management as a whole is also impractical. To solve the problem in a practical way, Quasar performs four parallel classifications on each application to evaluate the four main aspects of resource allocation and assignment: the impact of scale-up (amount of resources per server), the impact of scale-out (number of servers per workload), the impact of server configuration, and the impact of interference (which workloads can be colocated).

Third, Quasar *performs resource allocation and assignment jointly*. The classification results are used to determine the right amount and specific set of resources assigned to the workload. Hence, Quasar avoids overprovisioning workloads that are currently at low load and can compensate for increased interference or the unavailability of high-end servers by assigning fewer or lower-quality resources to them. Moreover, Quasar monitors performance throughout the workload’s execution. If performance deviates from the expressed constraints, Quasar reclassifies the workload and adjusts the allocation and/or assignment decisions to meet the performance constraints or minimize the resources used.

We have implemented and evaluated a prototype for Quasar managing a local 40-server cluster and a 200-node cluster of dedicated EC2 servers. We use a wide range of workloads including analytics frameworks (Hadoop, Storm, Spark), latency-critical and stateful services (memcached, Cassandra), and batch workloads. We compare Quasar to reservation-based resource allocation coupled with resource assignment based on load or similar classification techniques. Quasar improves server utilization at steady state by 47% on average at high load in the 200-server cluster, while also improving performance of individual workloads compared to the alternative schemes. We show that Quasar correctly determines the amount of resources needed by analytics and latency-critical workloads better than built-in schedulers of frameworks like Hadoop, or auto-scaling systems. It also selects assignments that take heterogeneity and interference into account so that throughput and latency constraints are closely met.

4.2 Motivation

4.2.1 Cluster Management Overview

A cluster management framework provides various services including security, fault tolerance, and monitoring. This work focuses on the two tasks most relevant to resource efficiency: resource allocation and resource assignment of incoming workloads. Previous work has mostly treated the two separately.

Resource allocation: Allocation refers to determining the amount of resources used by a workload: number of servers, number of cores and amount of memory and bandwidth resources per server. Managers like Mesos [139], Torque [259], and Omega [232] expect workloads to make resource reservations. Mesos processes these requests and, based on availability and fairness issues [114], makes resource offers to individual frameworks (e.g., Hadoop) that the framework can accept or reject. Dejavu identifies a few workload classes and reuses previous resource allocations for each class to minimize reallocation overheads [266]. CloudScale [240], PRESS [118], AGILE [203] and the work by Gmach et al. [117] perform online prediction of resource needs, often without a priori workload knowledge. Finally, auto-scaling systems such as Rightscale [224] automatically scale the number of physical or virtual instances used by webserving workloads to react to observed changes in server load.

Resource assignment: Assignment refers to selecting the specific resources that satisfy an allocation. The two biggest challenges of assignment are server heterogeneity and interference between colocated workloads [188, 199, 285], when servers are shared to improve utilization. The most closely related work to Quasar is Paragon [76]. Given a resource allocation for an unknown, incoming workload, Paragon uses classification techniques to quickly estimate the impact of heterogeneity and interference on performance. Paragon uses this information to assign each workload to server type(s) that provide the best performance and colocate workloads that do not interfere with each other. Nathuji et al. [200] developed a feedback-based scheme that tunes resource assignment to mitigate interference effects. Yang et al. developed an online scheme that detects memory pressure and finds colocations that avoid interference

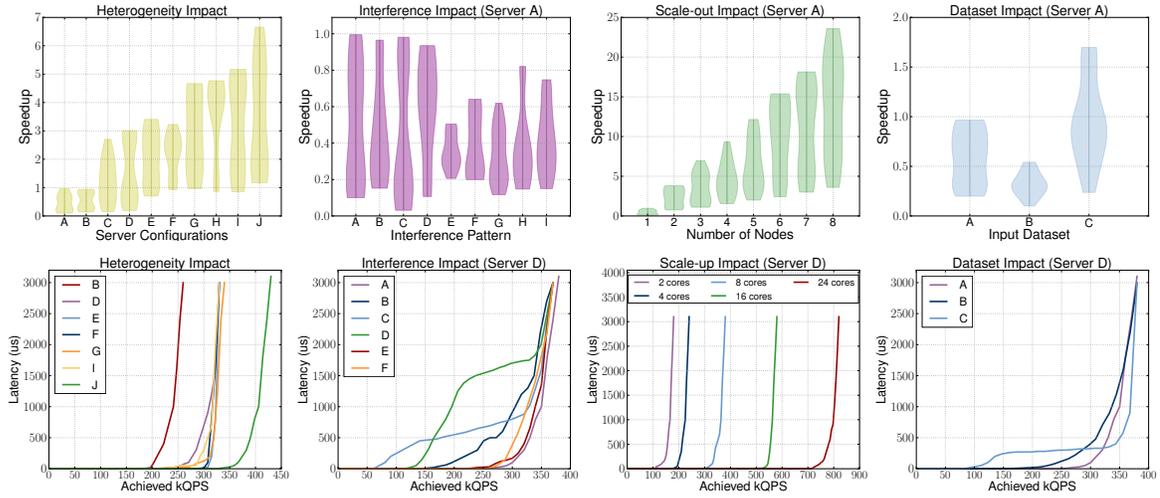


Figure 4.2: The impact of heterogeneity, interference, scale-out, scale-up, and dataset on the performance of Hadoop (top row) and memcached (bottom row). Server configurations, interference patterns, and datasets are summarized in Table 4.1. For Hadoop, the variability in the violin plots is due to scaling-up the resource allocations within a server (cores and/or memory).

on latency-sensitive workloads [285]. Similarly, DeepDive detects and manages interference between co-scheduled applications in a VM system [204]. Finally, CPI2 [296] throttles low-priority workloads that induce interference to important services. In terms of managing heterogeneity, Nathuji et al. [199] and Mars et al. [186] quantified its impact on conventional benchmarks and Google services and designed schemes to predict the most appropriate server type for each workload.

4.2.2 The Case for Coordinated Cluster Management

Despite the progress in cluster management technology, resource utilization is quite low in most private and public clouds (see Figure 4.1 and [63, 109, 177, 223, 265]). There are two major shortcomings current cluster managers have. First, it is particularly difficult for a user or workload to understand its resource needs and express them as a reservation. Second, resource allocation and assignment are fundamentally linked. An efficient allocation depends on the amount and type of resources available and the behavior of other workloads running on the cluster.

Figure 4.2 illustrates these issues by analyzing the impact of various allocations,

platforms	A	B	C	D	E	F	G	H	I	J
cores	2	4	8	8	8	8	12	12	16	24
memory(GB)	4	8	12	16	20	24	16	24	48	48
interference	A	B	C	D	E	F	G	H	I	
pattern	-	memory	LII \$	LL \$	disk I/O	network	L2 \$	CPU	prefetch	
input	A			B			C			
hadoop	netflix: 2.1GB			mahout: 10GB			wikipedia: 55GB			
memcached	100B reads			2KB reads			100B reads-100B writes			

Table 4.1: Server platforms (A - J), interference patterns (A - I) and input datasets (A - C) used for the analysis in Figure 4.2.

assignments, and workload aspects on two representative applications, one batch and one latency-critical: a large Hadoop job running a recommendation algorithm on the Netflix dataset [34] and a memcached service under a read-intensive load. For Hadoop, we report speedup over a single node of server configuration A using all available cores and memory. Server configurations, interference settings and datasets are summarized in Table 4.1. The variability in each violin plot is due to the different amounts of resources (cores and memory) allocated within each server. For memcached, we report the latency-throughput graphs. Real-world memcached deployments limit throughput to achieve 99th-percentile latencies between 0.2ms and 1ms.

The first row of Figure 4.2 illustrates the behavior of Hadoop. The heterogeneity graph shows that the choice of server configuration introduces performance variability of up to 7x, while the amount of resources allocated within each server introduces variability of up to 10x. The interference graph shows that for server A, depending on the amount of resources used, Hadoop may be insensitive to certain types of interference or slowdown by up to 10x. Similarly, the scale-out graph shows that depending on the amount of resources per server, scaling may be sublinear or super-linear. Finally, the dataset graph shows that the dataset complexity and size can have 3x impact on Hadoop’s performance. Note that in addition to high variability, the violin plots show that the probability distributions change significantly across different allocations. The results are similar for memcached, as shown in the second row of Figure 4.2. The position of the knee of the throughput-latency curve depends heavily on the type of server used (3x variability), the interference patterns (7x variability), the amount of resources used per server (8x variability), and workload characteristics

such as data size and read/write mixes (3x variability).

It is clear from Figure 4.2 that it is quite difficult for a user or workload to translate a performance goal to a resource reservation. To right-size an allocation, we need to understand how scale-out, scale-up, heterogeneity in the currently available servers, and interference from the currently running jobs affect a workload with a specific dataset. Hence, separating resource allocation and assignment through reservations is bound to be suboptimal, either in terms of resource efficiency or in terms of workload performance (see Figure 4.1). Similarly, performing first allocation and then assignment in two separate steps is also suboptimal. Cluster management must handle both tasks in an integrated manner.

4.3 Quasar

4.3.1 Overview

Quasar differs from previous work in three ways. First, it shifts away from resource reservations and adopts a *performance-centric* approach. Quasar exports a high-level interface that allows users or the schedulers integrated in some frameworks (e.g., Hadoop or Spark) to express the performance constraint the workload should meet. The interface differentiates across workload types. For latency-critical workloads, constraints are expressed as a queries per second (QPS) target and a latency QoS constraint. For distributed frameworks like Hadoop, the constraint is execution time. For single node, single-threaded or multi-threaded workloads the constraint is a low-level metric of instructions-per-second (IPS). Once performance constraints are specified, it is up to Quasar to find a resource allocation and assignment that satisfies them.

Second, Quasar uses fast classification techniques to quickly and accurately estimate the impact different resource allocation and resource assignment decisions have on workload performance. Upon admission, an incoming workload and dataset is profiled on a few servers for a short period of time (a few seconds up to a few minutes

- see Section 4.3.2). This limited profiling information is combined with information from the few workloads characterized offline and the many workloads that have been previously scheduled in the system using classification techniques. The result of classification is accurate estimates of application performance as we vary the type or number of servers, the amount of resources within a server, and the interference from other workloads. In other words, we estimate graphs similar to those shown in Figure 4.2. This classification-based approach eliminates the need for exhaustive online characterization and allows efficient scheduling of unknown or evolving workloads, or new datasets. Even with classification, exhaustively estimating performance for all allocation-assignment combinations would be infeasible. Instead, Quasar decomposes the problem to the four main components of allocation and assignment: resources per node and number of nodes for allocation, and server type and degree of interference for assignment. This dramatically reduces the complexity of the classification problem.

Third, Quasar uses the result of classification to jointly perform resource allocation and assignment, eliminating the inherent inefficiencies of performing allocation without knowing the assignment challenges. A greedy algorithm combines the result of the four independent classifications to select the number and specific set of resources that will meet (or get as close as possible to) the performance constraints. Quasar also monitors workload performance. If the constraint is not met at some point or resources are idling, either the workload changed (load or phase change), classification was incorrect, or the greedy scheme led to suboptimal results. In any case, Quasar adjusts the allocation and assignment if possible, or reclassifies and reschedules the workload from scratch.

Quasar uses similar classification techniques as those introduced in Paragon [76]. Paragon handles only resource assignment. Hence, its classification step can only characterize workloads with respect to heterogeneity (server type) and interference. In contrast, Quasar handles both resource allocation and assignment. Hence, its classification step also characterizes scale-out and scale-up issues for each workload.

Moreover, the space of allocations and assignments that Quasar must explore is significantly larger than the space of assignments explored by Paragon. Finally, Quasar introduces an interface for performance constraints in order to decouple user goals from resource allocation and assignment. In Section 4.6, we compare Quasar to Paragon coupled with current resource allocation approaches to showcase the advantages of Quasar.

4.3.2 Fast and Accurate Classification

Collaborative filtering techniques are often used in recommendation systems with extremely sparse inputs [219]. One of their most publicized uses was the Netflix Challenge [34], where techniques such as Singular Value Decomposition (SVD) and PQ-reconstruction [43, 161, 219, 281] were used to provide movie recommendations to users that had only rated a few movies themselves, by exploiting the large number of ratings from other users. The input to SVD in this case is a very sparse matrix A with users as rows, movies as columns and ratings as elements. SVD decomposes A to the product of the matrix of singular values Σ that represents similarity concepts in A , the matrix of left singular vectors U that represents correlation between rows of A and similarity concepts, and the matrix of right singular vectors V that represents the correlation between columns of A and similarity concepts ($A = U \cdot \Sigma \cdot V^T$). A similarity concept can be that users that liked “Lord of the Rings 1” also liked “Lord of the Rings 2”. PQ-reconstruction with Stochastic Gradient Descent (SGD), a simple latent-factor model [43, 281], uses Σ , U , and V to reconstruct the missing entries in A . Starting with the SVD output, P^T is initialized to ΣV^T and Q to U which provides an initial reconstruction of A . Subsequently, SGD iterates over all elements of the reconstructed matrix $R=Q \cdot P^T$ until convergence.

For each element r_{ui} of R :

$$\begin{aligned} \epsilon_{ui} &= r_{ui} - \mu - b_u - q_i \cdot p_u^T \\ q_i &\leftarrow q_i + \eta(\epsilon_{ui} p_u - \lambda q_i) \\ p_u &\leftarrow p_u + \eta(\epsilon_{ui} q_i - \lambda p_u) \end{aligned}$$

until $|\epsilon|_{L_2} = \sqrt{\sum_{u,i} |\epsilon_{ui}|^2}$ becomes marginal. η is the learning rate and λ the regularization factor of SGD and their values are determined empirically. In the above model, we also include the average rating μ and a user bias b_u that account for the divergence of specific users from the norm. Once the matrix is reconstructed, SVD is applied once again to generate movie recommendations by quantifying the correlation between new and existing users. The complexity of SVD is $O(\min(N^2M, M^2N))$, where M, N the dimensions of A , and the complexity of PQ-reconstruction with SGD is $O(N \cdot M)$.

In Paragon [76], collaborative filtering was used to quickly classify workloads with respect to interference and heterogeneity. A few applications are profiled exhaustively offline to derive their performance on different servers and with varying amounts of interference. An incoming application is profiled for one minute on two of the many server configurations, with and without interference in two shared resources. SVD and PQ-reconstruction are used to accurately estimate the performance of the workload on the remaining server configurations and with interference on the remaining types of resources. Paragon showed that collaborative filtering can quickly and accurately classify unknown applications with respect to tens of server configurations and tens of sources of interference.

The classification engine in Quasar extends the one in Paragon in two ways. First, it uses collaborative filtering to estimate the impact of resource scale-out (more servers) and scale-up (more resources per server) on application performance. These additional classifications are necessary for resource allocation. Second, it tailors classifications to different workload types. This is necessary because different types for workloads have different constraints and allocation knobs. For instance, in a web-server we can apply both scale-out and scale-up and we must monitor queries per second (QPS) and latency. For Hadoop, we can also configure workload parameters such as the number of mappers per node, heapsize, and compression. For a single-node workload, scaling up might be the only option while the metric of interest can be instructions per second. The performance constraints interface of Quasar allows users to specify the type of submitted applications.

Overall, Quasar classifies for scale-up, scale-out, heterogeneity, and interference.

The four classifications are done independently and in parallel to reduce complexity and overheads. The greedy scheduler combines information from all four. Because of the decomposition of the problem the matrix dimensions decrease, and classification becomes fast enough that it can be applied on every workload submission, even if the same workload is submitted multiple times with different datasets. Hence there is no need to classify for dataset sensitivity.

Scale-up classification: This classification explores how performance varies with the amount of resources used within a server. We currently focus on compute cores, memory and storage capacity. We will address network bandwidth in future work. We perform scale-up classification on the highest-end platform, which offers the largest number of scale-up options. When a workload is submitted, we profile it briefly with two randomly-selected scale-up allocations. The parameters and duration of profiling depend on workload type. Latency-critical services, like memcached are profiled for 5-10 seconds under live traffic, with two different core/thread counts and memory allocations (see the validation section for a sensitivity analysis on the number of profiling runs). For workloads like Hadoop, we profile a small subset (2-6) of map tasks with two different allocations and configurations of the most important framework parameters (e.g., mappers per node, JVM heapsize, block size, memory per task, replication factor, and compression). Profiling lasts until the map tasks reach at least 20% of completion, which is typically sufficient to estimate the job's completion time using its progress rate [289] and assuming uniform task duration [139]. Section 4.4.3 addresses the issue of non-uniform task duration distribution and stragglers. Finally, for stateful services like Cassandra [49], Quasar waits until the service's setup is complete before profiling the input load with the different allocations. This takes at most 3-5 minutes, which is tolerable for long-running services. Section 4.4.2 discusses how Quasar guarantees side-effect free application copies for profiling runs.

Profiling collects performance measurements in the format of each application's performance goal (e.g., expected completion time or QPS) and inserts them into a matrix A with workloads as rows and scale-up configurations as columns. A configuration includes compute, memory, and storage allocations or the values of the framework parameters for a workload like Hadoop. To constrain the number of columns, we

quantize the vectors to integer multiples of cores and blocks of memory and storage. This may result into somewhat suboptimal decisions, but the deviations are small in practice. Classification using SVD and PQ-reconstruction then derive the workload's performance across all scale-up allocations.

Scale-out classification: This type of classification is only applicable to workloads that can use multiple servers, such as distributed frameworks (e.g., Hadoop or Spark), stateless (e.g., webserving) or stateful (e.g., memcached or Cassandra) distributed services, and distributed computations (e.g., MPI jobs). Scale-out classification requires one more run in addition to single-node runs done for scale-up classification. To get consistent results, profiling is done with the same parameters as one of the scale-up runs (e.g., JVM heapsize) and the same application load. This produces two entries for matrix A , where rows are again workloads and columns are scale-out allocations (numbers of servers). Collaborative filtering then recovers the missing entries of performance across all node counts. Scale-out classification requires additional servers for profiling. To avoid increasing the classification overheads when the system is online, applications are only profiled on one to four nodes for scale-out classification. To accurately estimate the performance of incoming workloads for larger node counts, in offline mode, we have exhaustively profiled a small number of different workload types (20-30) against node counts 1 to 100. These runs provide the classification engine with dense information on workload behavior for larger node counts. This step does not need to repeat unless there are major changes in the cluster's hardware or application structure.

Heterogeneity classification: This classification requires one more profiling run on a different and randomly-chosen server type using the same workload parameters and for the same duration as a scale-up run. Collaborative filtering estimates workload performance across all other server types.

Interference classification: This classification quantifies the sensitivity of the workload to interference caused and tolerated in various shared resources, including the CPU, cache hierarchy, memory capacity and bandwidth, and storage and network bandwidth. This classification does not require an extra profiling run. Instead, it leverages the first copy of the scale-up classification to inject, one at a time, two

microbenchmarks that create contention in a specific shared resource [74]. Once the microbenchmark is injected, Quasar tunes up its intensity until the workload performance drops below an acceptable level of QoS (typically 5%). This point is recorded as the workload’s sensitivity to this type of interference in a new row in the corresponding matrix A . The columns of the matrix are the different sources of interference. Classification is then applied to derive the sensitivities to the remaining sources of interference. Once the profiling runs are complete the different types of classification reconstruct the missing entries and provide recommendations on efficient allocations and assignments for each workload. Classification typically takes a few msec even for thousands of applications and servers.

Multiple parallel versus single exhaustive classification: Classification is decomposed to the four components previously described for both accuracy and efficiency reasons. The alternative design would consist of a single classification that examines all combinations of resource allocations and resource assignments at the same time. Each row in this case is an incoming workload, and each column is an allocation-assignment vector. Exhaustive classification addresses pathological cases that the four simpler classifications estimate poorly. For example, if TCP incast occurs for a specific allocation, *only* on a specific server platform that is not used for profiling, its performance impact will not be identified by classification. Although these cases are rare, they can result in unexpected performance results. On the other hand, the exponential increase in the column count in the exhaustive scheme increases the time required to perform classification [281, 161, 219] (note that this occurs at every application arrival). Moreover, because the number of columns now exceeds the number of rows, classification accuracy decreases, as SVD finds fewer similarities with high confidence [213, 275, 112].

To address this issue without resorting to exhaustive classification, we introduce a simple feedback loop that updates the matrix entries when the performance measured at runtime deviates from the one estimated through classification. This loop addresses such misclassifications, and additionally assists with scaling to server counts that exceed the capabilities of profiling, i.e., more than 100 nodes.

Validation: Table A.4 summarizes a validation of the accuracy of the classification

<i>Default density constraint: 2 entries per row, per classification</i>												
Classification err.	scale-up			scale-out			heterogeneity			interference		
	avg	90 th	max	avg	90 th	max	avg	90 th	max	avg	90 th	max
Hadoop (10 Jobs)	5.2%	9.8%	11%	5.0%	14.5%	17%	4.1%	4.6%	5.0%	1.8%	5.1%	6%
Memcached (10)	6.3%	9.2%	11%	6.6%	10.5%	12%	5.2%	5.7%	6.5%	7.2%	9.1%	10%
Webserver (10)	8.0%	10.1%	13%	7.5%	11.6%	14%	4.1%	5.1%	5.2%	3.2%	8.1%	9%
Single-node (413)	4.0%	8.1%	9%	-	-	-	3.5%	6.9%	8.0%	4.4%	9.2%	10%

Table 4.2: Validation of Quasar’s classification engine. We present average, 90th percentile and maximum errors between estimated values and actual values obtained with detailed characterization.

<i>8 entries per row</i>				
exhaustive classification				
Classification error	avg	90 th	%ile	max
Hadoop (10 Jobs)	14.1%	15.8%		16%
Memcached (10)	14.1%	16.5%		18%
Webserver (10)	16.5%	17.6%		18%
Single-node (413)	11.6%	12.1%		13%

Table 4.3: We also compare the classification errors of the four parallel classification to a single, exhaustive classification that accounts for all combinations of resource allocation and resource assignment jointly.

engine in Quasar. We use a 40-server cluster and applications from Hadoop (10 data-mining jobs), latency-critical services (10 memcached jobs, and 10 Apache webserver loads), and 413 single-node benchmarks from SPEC, PARSEC, SPLASH-2, BioParallel, Minebench and SpecJbb. The memcached and webserving jobs differ in their query distribution, input dataset and/or incoming load. Hadoop jobs additionally differ in terms of the application logic. Details on the applications and systems can be found in Section 4.5. We show average, 90th percentile and maximum errors for each application and classification type. The errors show the deviation between estimated and measured performance or sensitivity to interference. On average, classification errors are less than 8% across all application types, while maximum errors are less than 17%, guaranteeing that the information that drives cluster management decisions is accurate. Table A.4 also shows the corresponding errors for the exhaustive classification. In this case, average errors are slightly higher, especially for applications arriving early in the system [213], however, the deviation between average and maximum errors is now lower, as the exhaustive classification can accurately predict performance for the pathological cases that the four parallel classifications miss.

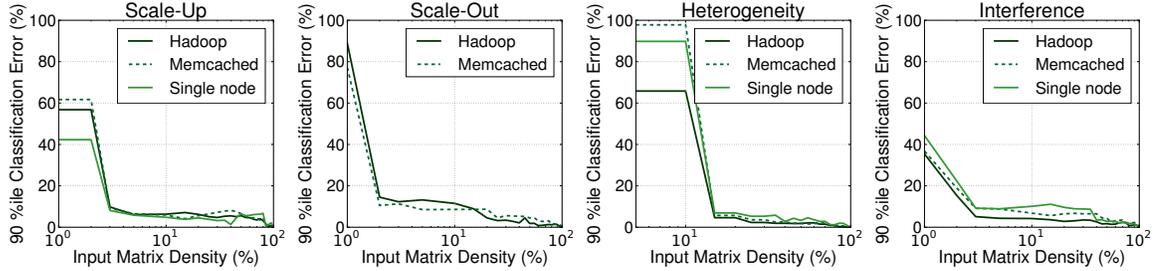


Figure 4.3: Sensitivity of classification accuracy to input matrix density constraints (Figure 4.3(a-d)).

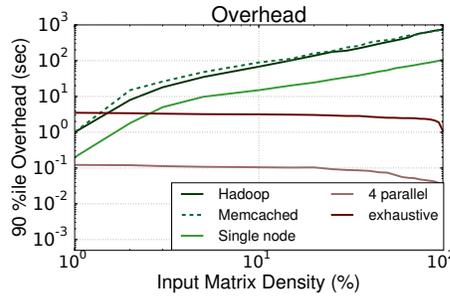


Figure 4.4: The profiling and decision overheads for different density constraints of the input matrices, assuming constant hardware resources for profiling.

We also validate the selected number of profiling runs, i.e., how classification accuracy changes with the density of the input matrices. Figure 4.3(a-d) shows how the 90th percentile of errors from classification changes as the density of the corresponding input matrix increases. For clarity, we omit the plots for Webserver, which has similar patterns to memcached. For all four classification types, a single profiling run per classification results in high errors. Two or more entries per input row result in decreased errors, although the benefits reach the point of diminishing returns after 4-5 entries. This behavior is consistent across application types, although the exact values of errors may differ. Unless otherwise specified, we use 2 entries per row in subsequent experiments. Figure 4.4 shows the overheads (profiling and classification) for the three application classes (Hadoop, memcached, single node) as input matrix density increases. Overheads are calculated with respect to the useful execution time for each workload. We assume that the hardware resources used towards profiling and classification are kept constant. Obviously as the number of profiling runs

increases the overheads increase significantly, without equally important accuracy improvements. The figure also shows the overheads from classification only (excluding profiling) for the four parallel classifications (*4 parallel*) and the exhaustive scheme (*exhaustive*). As expected, the increase in column count corresponds in an increase in decision time, often by two orders of magnitude.

4.3.3 Greedy Allocation and Assignment

The classification output is given to a greedy scheduler that jointly determines the amount, type, and exact set of allocated resources. The scheduler’s objective is to allocate the *least amount of resources needed to satisfy a workload’s performance target*. This greatly reduces the space the scheduler traverses, allowing it to examine higher quality resources first, as smaller quantities of them will meet the performance constraint. This approach also scales well to many servers.

The scheduler uses the classification output, to first rank the available servers by decreasing resource quality, i.e., high performing platforms with minimal interference first. Next, it sizes the allocation based on available resources until the performance constraint is met. For example, if a webserver must meet a throughput of 100K QPS with 10msec 99th percentile latency and the highest-ranked servers can achieve at most 20K QPS, the workload would need five servers to meet the constraints. If the number of highest-ranked servers available is not sufficient, the scheduler will also allocate lower-ranked servers and increase their number. The feedback between allocation and assignment ensures that the amount and quality of resources are accounted for jointly. When sizing the allocation, the algorithm first increases the per-node resources (scale-up) to better pack work in few servers, and then distributes the load across machines (scale-out). Nevertheless, alternative heuristics can be used based on the workload’s locality properties or to address fault tolerance concerns.

The greedy algorithm has $O(M \cdot \log M + S)$ complexity, where the first component accounts for the sorting overhead and the second for the examination of the top S servers, and in practice takes a few msec to determine an allocation/assignment even for systems with thousands of servers. Despite its greedy nature, we show in

Section 4.6 that the decision quality is quite high, leading to both high workload performance and high resource utilization. This is primarily due to the accuracy of the information available after classification. A potential source of inefficiency is that the scheduler allocates resources on a per-application basis in the order workloads arrive. Suboptimal assignments can be detected by sampling a few workloads (e.g., based on job priorities if they are available) and adjusting their assignment later on as resources become available when other workloads terminate. Finally, the scheduler employs admission control to prevent oversubscription when insufficient resources are available.

4.3.4 Putting it All Together

Figure 4.5 shows the different steps of cluster management in Quasar. Upon arrival of a workload, Quasar collects profiling data for scale-out and scale-up allocations, heterogeneity, and interference. This requires up to four profiling runs that happen in parallel. All profiling copies are sandboxed (as explained in Section 4.4.2), the two platforms used are A and B (two nodes of A are used for the scale-out classification) and each profiling type produces two points in the corresponding speedup graph of the workload. The profiling runs happen with the actual dataset of the workload. The total profiling overhead depends on the workload type and is less than 5 min in all cases we examined. For non-stateful services, e.g., small batch workloads that are a large fraction of DC workloads [232], the complete profiling takes 10-15 seconds. Note that for stateful services, e.g., Cassandra, where setup is necessary, it only affects *one* of the profiling runs. Once the service is warmed-up, subsequent profiling only requires a few seconds to complete. Once the profiling results are available, classification provides the full workload characterization (speedup graph). Next, the greedy scheduler assigns specific servers to the workload. Overall, Quasar’s overheads are quite low even for short-running applications (batch, analytics) or long running online services.

Quasar maintains per-workload and per-server state. Per-workload state includes

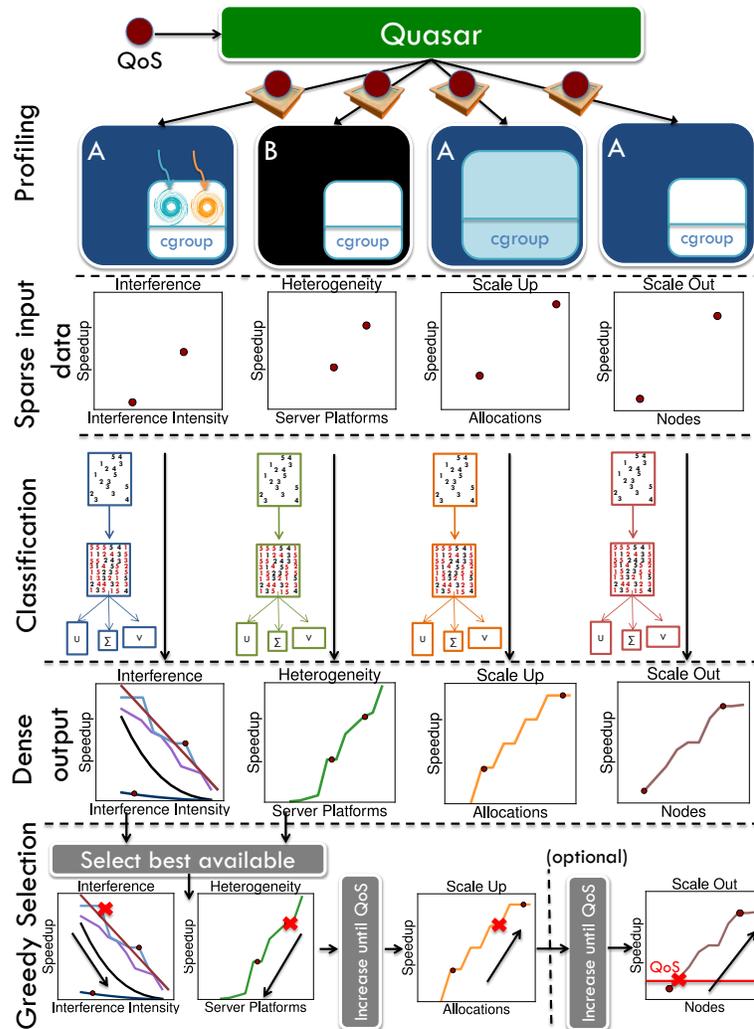


Figure 4.5: The steps for cluster management with Quasar. Starting from the top, short runs using sandboxed workload copies produce the initial profiling signal that classification techniques expand to information about relationship between performance and scale-up, scale-out, heterogeneity, and interference. Finally, the greedy scheduler uses the classification output to find the number and type of resources that maximize utilization and application performance.

the classification output. For a cluster with 10 server types and 10 sources of interference, we need roughly 256 bytes per workload. The per-server state includes information on scheduled applications and their cumulative resource interference, roughly 128B in total. The per-server state is updated on each workload assignment. Quasar also needs some storage for the intermediate classification results and for server ranking during assignment. Overall, state overheads are marginal and scale linearly with the number of workloads and servers. In our experiments, a single server was sufficient to handle the total state and computation of cluster management. Additional servers can be used for fault-tolerance.

4.4 Implementation

We implemented a prototype for Quasar in about 6KLOC of C, C++, and Python. It runs on Linux and OS X and currently supports applications written in C/C++, Java, and Python. The API includes functions to express the performance constraints and type of submitted workloads, and functions to check job status, revoke it, or update the constraints. We have used Quasar to manage analytics frameworks such as Hadoop, Storm, and Spark, latency-critical services such as NoSQL workloads, and conventional single-node workloads. There was no need to change any applications or frameworks. The framework-specific code in Quasar is 100-600 LOC per framework. In the future, we plan to merge the Quasar classification and scheduling algorithms in a cluster management framework like OpenStack or Mesos.

4.4.1 Dynamic Adaptation

Some workloads change behavior during their runtime, either due to phase changes or due to variation in user traffic. Quasar detects such changes and adjusts resource allocation and/or assignment to preserve the performance constraints.

Phase detection: Quasar continuously monitors the performance of all active workloads in the cluster. If a workload runs below its performance constraint, it either went through a phase change or was incorrectly classified or assigned. In

any case, Quasar reclassifies the application at its current state and adjusts its resources as needed (see discussion below). We also proactively test for phase changes and misclassifications/misscheduling by periodically sampling a few active workloads and injecting interfering microbenchmarks to them. This enables partial interference classification in place. If there is a significant change compared to the original classification results, Quasar signals a phase change. Proactive detection is particularly useful for long-running workloads that may affect colocated workloads when entering a phase change. We have validated the phase detection schemes with workloads from SPECCPU2006, PARSEC, Hadoop and memcached. With the reactive-only scheme, Quasar detects 94% of phase changes. By sampling 20% of active workloads every 10 minutes, we detect 78% of changes proactively with 8% probability of false positives.

Allocation adjustment: Once the phase has been detected or load increases significantly for a user-facing workload, Quasar changes the allocation to provide more resources or reclaim unused resources. Quasar adjusts allocations in a conservative manner. It first scales up or down the resources given to the workload in each of the servers it currently occupies. If needed, best-effort (low priority) workloads are evicted from these servers. If possible, a scale-up adjustment is the simplest option as it typically requires no state migration. If scale-up is not possible or cannot address the performance needs, scale-out and/or migration to other servers is used. For stateless services (e.g., adding/removing workers to Hadoop or scaling a webserver), scale-out is straight-forward. For stateful workloads, migration and scale-out can be expensive. If the application is organized in microshards [68], Quasar will migrate a fraction of the load from each server to add capacity at minimum overhead. At the moment, Quasar does not employ load prediction for user-facing services [118, 203]. In future work, we will use such predictors as an additional signal to trigger adjustments for user-facing workloads.

4.4.2 Side Effect Free Profiling

To acquire the profiling data needed for classification, we must launch multiple copies of the incoming application. This may cause inconsistencies with intermediate results,

duplicate entries in databases, or data corruption on file systems. To eliminate such issues, Quasar uses sandboxing for the training copies during profiling. We use *Linux containers* [26] with `chroot` to sandbox profiling runs and create a copy-on-write filesystem snapshot so that files (including framework libraries) can be read and written as usual [291]. Containers enable full control over how training runs interact with the rest of the system, including limiting resource usage through *cgroups*. Using virtual machines (VMs) for the same purpose is also possible [204, 269, 270, 283], but we chose containers as they incur lower overheads for launching.

4.4.3 Stragglers

In frameworks like Hadoop or Spark, individual tasks may take much longer to complete for reasons that range from poor work partitioning to network interference and machine instability [16]. These straggling tasks are typically identified and relaunched by the framework to ensure timely job completion [7, 15, 16, 69, 108, 176, 289]. We improve straggler detection in Hadoop in the following manner. Quasar calls the `TaskTracker` API in Hadoop and checks for underperforming tasks (at least 50% slower than the median). Straggling tasks are typically stalling in specific resources, which would alter the original interference profile. To detect this, Quasar injects two contentious microbenchmarks in the corresponding servers and reclassifies the underperforming tasks with respect to interference caused and tolerated. If the results of the in-place classification differ from the original by more than 20%, we signal the task as a straggler and notify the Hadoop JobTracker to relaunch it on a newly assigned server. This allows Quasar to detect stragglers 19% earlier than Hadoop, and 8% earlier than LATE [289] for the Hadoop applications described in the first scenario in Section 4.5.

4.4.4 Discussion

Cost target: Apart from a performance target, a user could also specify a cost constraint, priorities, and utility functions for a workload [238]. These can either serve as a limit for resource allocation or to prioritize allocations during very high

load.

Resource partitioning: Quasar does not explicitly partition hardware resources. Instead, it reduces interference by colocating workloads that do not contend on the shared resources. Resource partitioning is orthogonal. If mechanisms like cache partitioning or rate limiting at the NIC are used, interference can be reduced and more workload colocations will be possible using Quasar. In that case, Quasar will have to determine the settings for partitioning mechanisms, in the same way it determines the number of cores to use for each workload. We will consider these issues in future work.

Fault tolerance: We use master-slave mirroring to provide fault-tolerance for the server that runs the Quasar scheduler. All system state (list of active applications, allocations, QoS guarantees) is continuously replicated and can be used by hot-standby masters. Quasar can also leverage frameworks like ZooKeeper [21] for more scalable schemes with multiple active schedulers. Quasar does not explicitly add to the fault tolerance of frameworks like MapReduce. In the event of a failure, the cluster manager relies on the individual frameworks to recover missing worker data. Our current resource assignment does not account for fault zones. However, this is a straight forward extension for the greedy algorithm.

4.5 Methodology

Clusters: We evaluated Quasar on a 40-server local cluster and a 200-server cluster on EC2. The ten platforms of the local cluster range from dual core Atom boards to dual socket 24 core Xeon servers with 48GB of RAM. The EC2 cluster has 14 server types ranging from small to x-large instances. All servers are dedicated and managed only by Quasar, i.e., there is no interference from external workloads.

The following paragraphs summarize the workload scenarios used to evaluate Quasar. Scenarios include batch and latency-critical workloads and progressively evaluate different aspects of allocation and assignment. Unless otherwise specified experiments are run 7 times for consistency and we report the average and standard deviation.

Single Batch Job: Analytics frameworks like Hadoop [134], Storm [252], and Spark [288] are large consumers of resources on private and public clouds. Such frameworks have individual schedulers that set the various framework parameters (e.g., mappers per node and block size) and determine resource allocation (number of servers used). The allocations made by each scheduler are suboptimal for two reasons. First, the scheduler does not have full understanding of the complexity of the submitted job and dataset. Second, the scheduler is not aware of the details of available servers (e.g., heterogeneity), resulting in undersized or overprovisioned allocations. In this first scenario, a single Hadoop job is running at a time on the small cluster. This simple scenario allows us to compare the resource allocation selected by Hadoop to the allocation/assignment of Quasar on a single job basis. We use ten Hadoop jobs from the Mahout library [184] that represent data mining and machine learning analyses. The input datasets vary between 1 and 900GB. Note that there is no workload co-location in this scenario.

Multiple Batch Jobs: The second scenario represents a realistic setup for batch processing clusters. The cluster is shared between jobs from multiple analytics frameworks (Hadoop, Storm, and Spark). We use 16 Hadoop applications running on top of the Mahout library, four workloads for real-time text and image processing in Storm, and four workloads for logical regression, text processing and machine learning in Spark. These jobs arrive in the cluster with 5 sec inter-arrival times. Apart from the analytics jobs, a number of single-server jobs are submitted to the cluster. We use workloads from SPECCPU2006, PARSEC [40], SPLASH-2 [282], BioParallel [150], Minebench [198] and 350 multiprogrammed 4-app mixes from SPEC [228]. These single-server workloads arrive with 1 second inter-arrival times and are treated as best-effort (low priority) load that fills any cluster capacity unused by analytics jobs. There are not guarantees on performance of best-effort tasks, which may be migrated or killed at any point to provide resources for analytics tasks.

We compare Quasar to allocations done by the frameworks themselves (Hadoop, Spark, Storm schedulers) and assignments by a least-loaded scheduler that accounts for core and memory use but not heterogeneity or interference.

Low-Latency Service: Latency-critical services are also major tenants in cloud

facilities. We constructed a webserving scenario using the HotCRP conference management system [144], which includes the Apache webserver, application logic in PHP, and data stored in MySQL. The front- and back-end run on the same machine, and the installation is replicated across several machines. The database is kept purposefully small (5GB) so that it is cached in memory and emphasis is placed on compute, cache, memory and networking issues, and not on disk performance. HotCRP traffic includes requests to fill in paper abstracts, update author information, and upload or read papers. Apart from throughput constraints, HotCRP requires a 100msec per-request latency.

We use three traffic scenarios: flat, fluctuating, and large spike. Apart from satisfying HotCRP constraints, we want to use any remaining cluster capacity for single-node, best-effort tasks (see description in previous scenario). We compare Quasar to a system that uses an auto-scaling approach to scale HotCRP between 1 and 8 servers based on the observed load of the servers used [24]. Auto-scale allocates an additional, least-loaded server for HotCRP when current load exceeds 70% [25] and redirects a fair share of the traffic to the new server instance. Load balancing happens on the workload generator side. Best-effort jobs are assigned by a least-loaded (LL) scheduler. Quasar deals with load changes in HotCRP by either scaling-up existing allocations or scaling-out (more servers) based on how the two affect performance.

Stateful Latency-Critical Services: This scenario extends the one above in two ways. First, there are multiple low-latency services. Second, these services involve significant volumes of state. Specifically, we examine the deployment of memory-based memcached [107] and disk-based Cassandra [49], two latency-critical NoSQL services. Memcached (1TB state) is presented with load that fluctuates following a diurnal pattern with maximum aggregate throughput target of 2.4M QPS and a 200usec latency constraint. The disk-bound Cassandra (4TB state) has a lower load of 60K QPS of maximum aggregate throughput and a 30 msec latency constraint. Any cluster capacity unused by the two services is utilized for best-effort workloads which are submitted with 10sec inter-arrival times. To show the fluctuation of utilization with load, and since scaling now involves state migration, this scenario runs over 24 hours and is repeated 3 times for consistency. Similarly to the previous scenario, we compare

Quasar with the auto-scaling approach and measure performance (throughput and latency) for the two services and overall resource utilization. Scale-out in this case involves migrating one (64MB) or more microshards to a new instance, which typically takes a few msec.

Large-Scale Cloud Provider: Finally, we bring everything together in a general case where 1200 workloads of all types (analytics batch, latency-critical, and single-server jobs) are submitted in random order to a 200-node cluster of dedicated EC2 servers with 1 sec inter-arrival time. All applications have the same priority and no workload is considered best-effort (i.e., all paying customers have equal importance). The scenario is designed to use almost all system cores at steady-state, without causing oversubscription, under ideal resource allocation. We do, however, employ admission control to prevent machine oversubscription, when allocation is imperfect [72]. Wait time due to admission control counts towards scheduling overheads. Quasar handles allocation and assignment for all workloads. For comparison, we use an auto-scale approach for resource allocation of latency-critical workloads. For frameworks like Hadoop and Storm, the framework estimates its resource needs and we treat that as a reservation. For resource assignment, we use two schedulers: a least-loaded scheduler that simply accounts for core and memory availability and Paragon that, given a resource allocation, can do heterogeneity- and interference-aware assignment. The latter allows us to demonstrate the benefits of jointly solving allocation and assignment over separate (although optimized) treatment of the two.

4.6 Evaluation

4.6.1 Single Batch Job

Performance: Figure 4.6 shows the reduction in execution time of ten Hadoop jobs when resources are allocated by Quasar instead of Hadoop itself. We account for all overheads, including classification and scheduling. Quasar improves performance for all jobs by an average of 29% and up to 58%. This is significant given that these

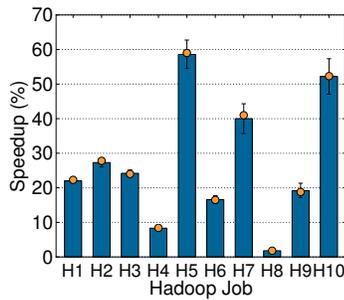


Figure 4.6: Performance of the ten Hadoop jobs with Quasar.

Parameter	Quasar	Hadoop
Block size	64MB	64MB
Compression rate/algorithm	7.6(gzip)	5.1(lzo)
Heapsize	0.75GB	1GB
Replication factor	2	2
Mappers per node	12	8
Server type	E-F	A-E

Figure 4.7: Parameter settings for Hadoop job H8 by Quasar and the default Hadoop scheduler.

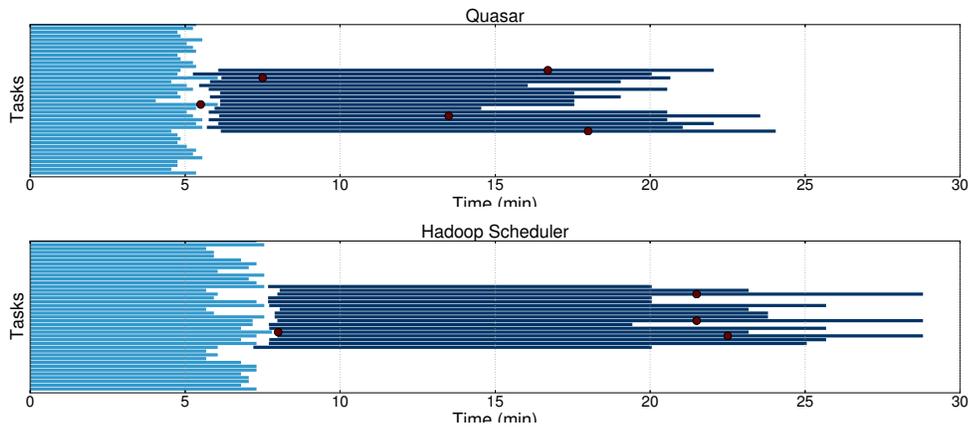


Figure 4.8: Straggler detection by Quasar versus the default Hadoop JobTracker.

Hadoop jobs take two to twenty hours to complete. The yellow dots show the execution time improvement needed to meet the performance target the job specified at submission. Targets are set to the best performance achieved after a parameter sweep on the different server platforms. Quasar achieves performance within 5.8% of the constraint on average, leveraging the information of how resource allocation and assignment impact performance. When resources are allocated by Hadoop, performance deviates from the target by 23% on average.

Efficiency: Table 4.7 shows the different parameter settings selected by Quasar and by Hadoop for the H8 Hadoop job, a recommendation system that uses Mahout with a 20GB dataset [184]. Apart from the block size and replication factor, the two frameworks set job parameters differently. Quasar detects that interference between mappers is low and increases the mappers per node to 12. Similarly, it detects that

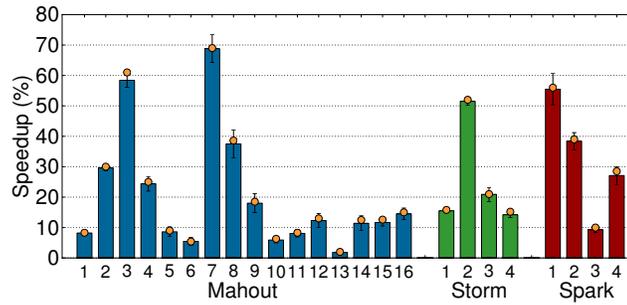


Figure 4.9: Performance speedup for the Hadoop, Storm and Spark jobs with Quasar.

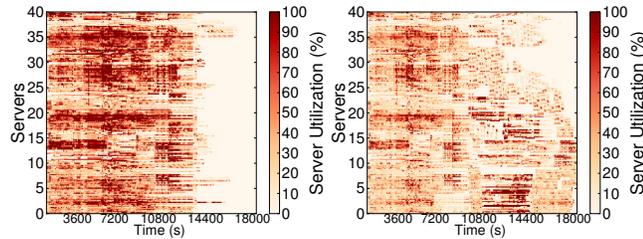


Figure 4.10: Cluster utilization with Quasar (left) and the framework schedulers (right). Darker colors correspond to higher CPU utilizations.

heap size is not critical for this job and reduces its size, freeing resources for other workloads. Moreover, Quasar allocates tasks to the two most suitable server types (E and F), while Hadoop chooses from all available server types.

4.6.2 Multiple Batch Frameworks

Performance: Figure 4.9 shows the reduction in execution times for Hadoop, Storm, and Spark jobs when Quasar manages resource allocation and assignment. On average, performance improves by 27% and comes within 5.3% of the provided constraint, a significant improvement over the baseline. Apart from sizing and configuring jobs better, Quasar can aggressively colocate them. For example, it can detect when two memory-intensive Storm and Spark jobs interfere and when they can efficiently share a system. Quasar allows the remaining cluster capacity to be used for best-effort jobs without disturbing the primary jobs because it is interference-aware. Best-effort jobs come within 7.8% on average of the peak performance each job could achieve if it was running alone on the highest performing server type.

Utilization: Figure 4.10 shows the per-server CPU utilization (average across

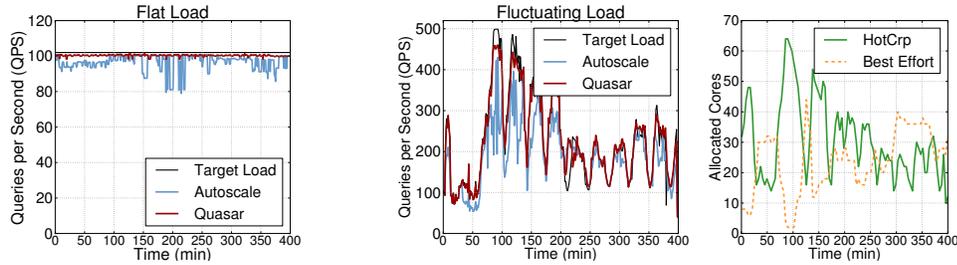


Figure 4.11: Throughput for HotCRP under (a) flat input load, and (b) fluctuating load. Figure 4.12(c) shows the core allocation in Quasar for the fluctuating load.

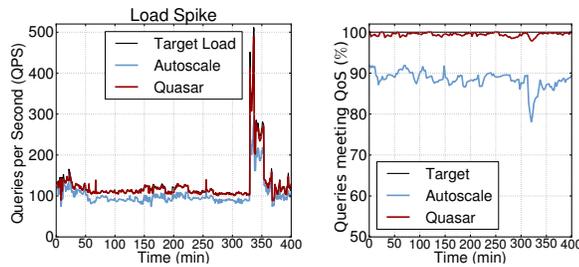


Figure 4.12: Throughput for HotCRP under (a) load with spikes. Figure 4.12(b) shows the fraction of queries meeting the latency constraint for the load with spikes.

all cores) over time in the form of a heatmap. Utilization is sampled every 5 sec. In addition to improving individual job performance, Quasar increases utilization, achieving 62% on average versus 34% with the individual framework schedulers (right heatmap). Because performance is now higher the whole experiment completes faster. Workloads after $t = 14400$ are mostly best-effort jobs that take longer than the main analytics workloads to complete.

4.6.3 Low-Latency Service

Performance: Figure 4.11a shows the aggregate throughput for HotCRP achieved with Quasar and the auto-scaling system when the input traffic is flat. While the absolute differences are small, it is important to note that the auto-scaling manager causes frequent QPS drops due to interference from best-effort workloads using idling resources. With Quasar, HotCRP runs undisturbed and the best-effort jobs achieve runtimes within 5% of minimum, while with auto-scale, they achieve runtimes within 24% of minimum. When traffic varies (Figure 4.11b), Quasar tracks target QPS

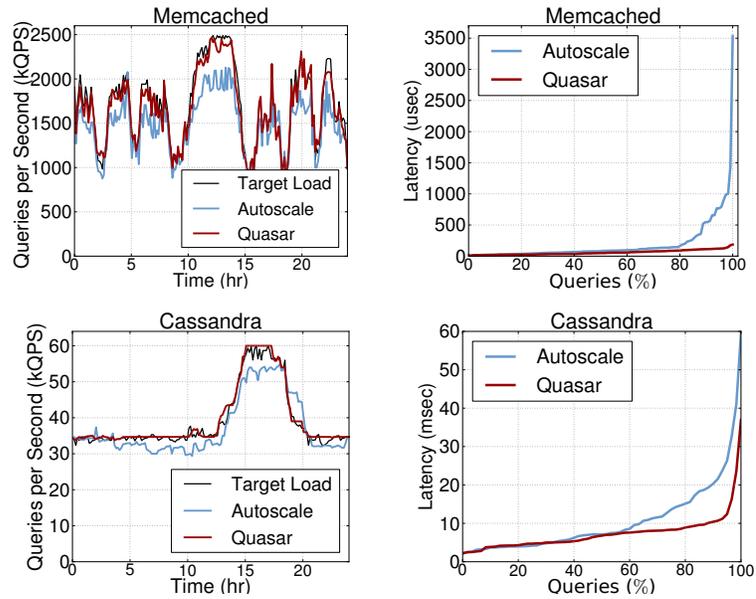


Figure 4.13: Throughput and latency for memcached and Cassandra in a cluster managed by Quasar or an auto-scaling system.

closely, while autoscale provides 18% lower QPS on average, both due to interference and suboptimal scale-up configuration. Quasar’s smooth behavior is due to the use of both scale-out and scale-up to best meet the new QPS target, leaving the highest number of cores possible for best-effort jobs (Figure 4.11c). For the load with the sharp spike, Quasar tracks QPS within 4% on average (Figure 4.12a) and meets the latency QoS for nearly all requests (Figure 4.12b). When the spike arrives, Quasar first scales up each existing allocation, and then only uses two extra servers of suitable type to handle remaining traffic. The auto-scaling system observes the load increase when the spike arrives and allocates four more servers. Due to the higher latency of scale-out and the fact that auto-scaling is not aware of heterogeneity or interference, it fails to meet the latency guarantees for over 20% of requests around the spike arrival.

4.6.4 Stateful Latency-Critical Services

Performance: Figure 4.13 shows the throughput of memcached and Cassandra over time and the distribution of query latencies. Quasar tracks throughput targets closely for both services, while the auto-scaling manager degrades throughput by 24% and

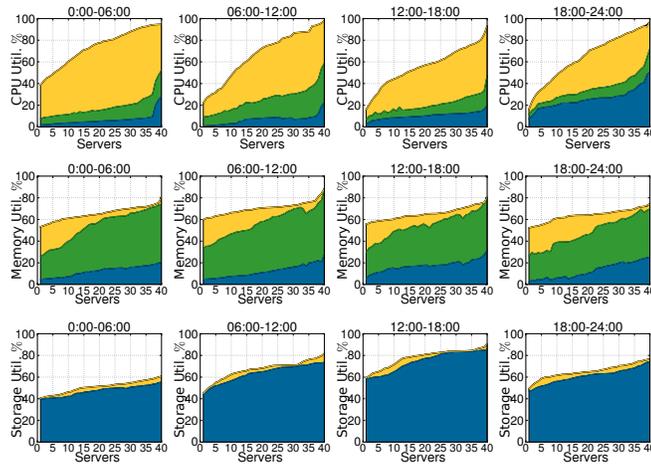


Figure 4.14: Average resource usage across all servers for four 6-hour snapshots. The cluster is running memcached (green), Cassandra (blue), and best-effort tasks (yellow) and is managed by Quasar.

12% on average for memcached and Cassandra respectively. The differences in latency are larger between the two managers. Quasar meets latency QoS for memcached for 98.8% of requests, while auto-scaling only for 80% of requests. For Cassandra, Quasar meets the latency QoS for 98.6% of requests, while the auto-scaling for 93% of requests. Memcached is memory-based and has an aggressive latency QoS, making it more sensitive to suboptimal resource allocation and assignment on a shared cluster.

Utilization: Figure 4.14 shows the utilization of CPU, memory capacity, and disk bandwidth across the cluster servers when managed by Quasar over 24h. Each column is a snapshot of average utilization over 6 hours. Since memcached and Cassandra have low CPU requirements, excluding the period 18:00-24:00 when Cassandra performs garbage collection, most of the CPU capacity is allocated to best-effort jobs. The number of best-effort jobs varies over time because the exact load of memcached and Cassandra changes. Most memory is used to satisfy the requirements of memcached, with small amounts needed for Cassandra and best-effort jobs. Cassandra is the nearly exclusive user of disk I/O. Some servers do not exceed 40-50% utilization for most of the experiment’s duration. These are low-end machines, for which higher utilization dramatically increases the probability of violating QoS constraints for latency-critical services. In general, the cluster utilization is significantly higher

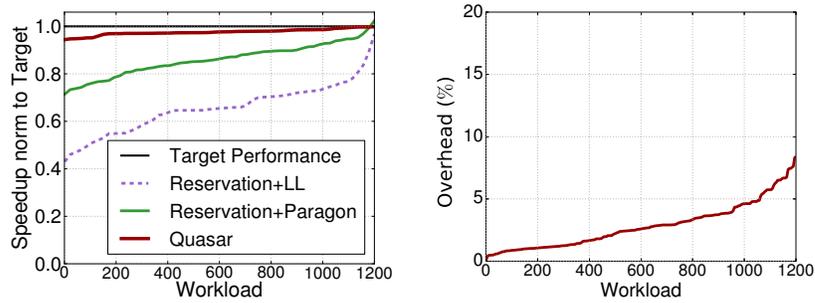


Figure 4.15: (a) Performance across the 1200 workloads on 200 EC2 servers with Quasar, the Reservation+Paragon, and the Reservation+LL scheduler. Figure 4.15(b) shows the cluster management overheads.

than if each service was running in dedicated machines.

4.6.5 Large-Scale Cloud Provider

Performance: Figures 4.15 and 4.16 present the overall evaluation of Quasar managing a 200-node cluster running all previously-discussed types of workloads. We compare to resource allocation based on reservations (e.g., expressed by the Hadoop scheduler or an auto-scaling system) and resource assignment on least-loaded machines (LL) or based on the interference and heterogeneity-aware Paragon. Figure 4.15a shows the performance of the 1,200 workloads ordered from worst- to best-performing, normalized to their performance target. Quasar achieves 98% of the target on average, while the reservation-based system with Paragon achieves 83%. This shows the need to perform allocation and assignment together; the intelligent resource assignment by Paragon is not sufficient. Using reservations and LL assignment performs quite poorly, only achieving 62% of the target on average.

Cluster management overheads: Figure 4.15b shows the cluster management overheads across the 1200 workloads. For most applications the overheads of Quasar from profiling, classification, greedy selection and adaptation are low, 4.1% of execution time on average. For short-lived batch workloads, overheads are up to 9%. The overheads are negligible for any long-running service, and even for jobs lasting a few seconds, they only induce single-digit increases in execution time. In contrast with reservation+LL, Quasar does not introduce any wait time overheads due to

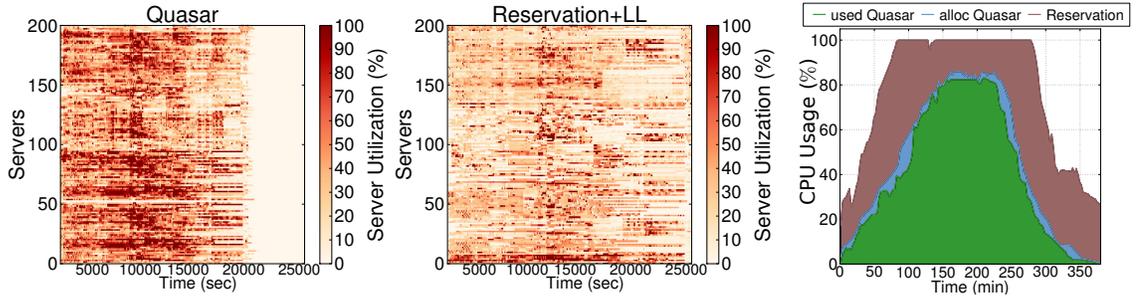


Figure 4.16: Cluster utilization for 1200 workloads on 200 EC2 servers with (a) Quasar and (b) the reservation+LL system. Figure 4.16(c) shows the allocated versus used resources for Quasar and allocated for reservation+LL.

oversubscription.

Utilization: Figures 4.16a-b show the per-server CPU utilization throughout the scenario’s execution for Quasar and the reservation+LL system. Average utilization is 62% with Quasar, while meeting performance constraints for both batch and latency-critical workloads. The reservation+LL manager achieves average utilization of 15%, 47% lower than Quasar. Figure 4.16c shows the allocated and used resources for Quasar compared to the resources reserved by the reservation+LL manager over time. Overprovisioning with Quasar is low, with the difference between allocated and used being roughly 10%. This is significantly lower than the resources reserved by the reservation-based manager, which exceed the capacity of the cluster during most of the scenario. Because Quasar has detailed information on how different allocations/assignments affect performance, it can rightsize the allocations more aggressively, while meeting the performance constraints without QoS violations.

4.7 Conclusions

We have presented Quasar, a cluster management system that performs coordinated resource allocation and assignment. Quasar moves away from the reservation-based standard for cluster management. Instead of users requesting raw resources, they specify a performance target the application should meet and let the manager size resource allocations appropriately. Quasar leverages robust classification techniques

to quickly analyze the impact of resource allocation (scale-up and scale-out), resource type (heterogeneity), and interference on performance. A greedy algorithm uses this information to allocate the least amount of resources necessary to meet performance constraints. Quasar currently supports distributed analytics frameworks, web-serving applications, NoSQL datastores, and single-node batch workloads. We evaluated Quasar over a variety of workload scenarios and compared it to reservation/auto-scaling-based resource allocation systems and schedulers that use similar classification techniques for resource assignment (but not resource allocation). We showed that Quasar improves aggregate cluster utilization and individual application performance.

Chapter 5

iBench: Quantifying Interference in Datacenter Workloads

5.1 Introduction

In the previous two chapters we discussed the challenges that interference in shared resources poses to both performance and efficiency. We also presented a fast technique to estimate the sensitivity of a new application to different types of interference. In this chapter we expand on this analysis, and present a new benchmark suite that enables this characterization.

Resource requirements vary widely across application types. Figure 5.1 for example, shows the memory capacity and memory bandwidth requirements of a wide set of application types, including single-threaded (ST) and multi-threaded (MT) benchmark suites such as SPEC CPU2006, PARSEC [40], SPLASH-2 [282], BioParallel [150] and MineBench [198], multiprogrammed (MP) mixes of these workloads, distributed batch (Hadoop) and latency-critical (memcached) applications, as well as traditional relational database workloads (MySQL). Capacity and bandwidth demands are normalized to the provisioned system values. The size of each bubble corresponds to the size of each job (number of tasks or clients). It becomes obvious that even when looking only at memory requirements, demands vary widely. Therefore, understanding the sensitivity workloads have to contention is critical towards reducing and managing

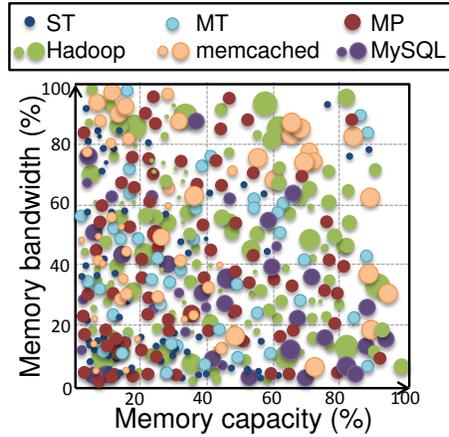


Figure 5.1: Pressure in memory capacity and memory bandwidth from a wide set of applications, as measured by iBench. The bubble size is proportional to the number of tasks (for Hadoop) or clients (for memcached) of the corresponding application.

interference in a way that enables QoS-aware operation at high utilization.

Previous work has shown the importance of accounting for interference in data-center scheduling [76, 187] and has developed hardware and software mechanisms to minimize interference effects. Mars et al. [187] show that ignoring the interference characteristics of large cloud applications in the memory subsystem can cause significant performance degradations that violate the workloads' QoS constraints. Typically, determining the interference profile of a workload involves either retroactively observing which co-scheduled applications contend in shared resources and annotating the offending workloads [296] or profiling the workload against a carefully-crafted benchmark that puts pressure on a specific shared resource [76, 187]. The disadvantage of the first approach is that interference is determined after performance degradation has occurred, and, currently, requires manual annotation of contending workloads. The second approach is less invasive, enables interference detection before this reflects into performance degradation, but requires effort in designing targeted benchmarks that put pressure on specific resources. Currently, there is no open-source benchmark suite that enables fast characterization of the interference an application tolerates and causes in various subsystems.

In this chapter we present iBench, a novel benchmark suite that helps quantify

the sensitivity of datacenter (and conventional) applications to interference. iBench consists of a set of carefully-crafted benchmarks that generate contention of tunable intensity in various shared resources which include the core, the cache and memory hierarchy, and the storage and networking subsystems. iBench workloads are called SoIs (sources of interference). Injecting an SoI in a machine hosting an application identifies the interference that application can tolerate in the corresponding shared resource before it violates its QoS, and the interference it itself creates in the same resource. We validate iBench against a set of datacenter applications that range from distributed frameworks such as Hadoop [134], latency-critical online services like memcached [107] and conventional single-threaded, multithreaded and multiprogrammed single-node applications, and verify the accuracy and consistency of the interference measurements.

We have used iBench in various system studies, and specifically in this work we show that it improves decision quality in four use cases that extend to cloud and chip multiprocessor (CMP) systems and span hardware and software challenges. First, we use the benchmark suite to quantify the interference sensitivity of a large set of applications resembling a cloud provider mix and use this information to make resource-efficient scheduling decisions. Second, we use iBench to guide the hardware configuration of datacenter servers, such that the system is appropriately provisioned to tolerate the pressure workloads put in different resources. Third, we move the interference characterization one step in advance and use it to guide application software development, before the workload’s full deployment. iBench here is used to determine the resources where an application induces contention, and to assist the software developer to design more resource-efficient code. Given the speed of interference characterization, using iBench significantly accelerates the iterative testing process of application software. Finally, we show that iBench is applicable to studies outside datacenters and use the interference characterization to guide scheduling decisions in a large-scale heterogeneous CMP. Note that in this case characterization needs to also account for the different core designs, while being lightweight and transparent to the workload. In all cases, using iBench significantly improves the system’s ability to preserve QoS guarantees in a resource-efficient manner. Specifically, scheduling in

a datacenter using iBench preserves performance for the majority of workloads, while significantly increasing utilization, by 42%. Also, by revising code regions, based on indications from iBench, we managed to reduce the application footprint of a large, data mining application by 49%, *while* speeding up the workload by 35%.

5.2 Related Work

DC benchmark suites: A major roadblock when studying DC applications is the unavailability of representative workloads and input loads. Given this challenge, there is extensive work on characterization and modeling of DC applications [23, 85, 84, 87, 86, 88, 83, 151, 234] that leads to generated workloads with characteristics that closely resemble those of the original application. The generated workloads can then be used in system studies without the limitation of needing access to real DC workloads. While this is a viable approach in some cases, modeling has limitations; there are workload aspects that are not captured in the model to preserve simplicity. However omitting these aspects can cause the generated workload to deviate from its expected behavior. Additionally, modeling is more applicable to large, long-running applications that can be characterized in detail to provide some input to the model, but is less beneficial in systems like Amazon’s EC2 or Windows Azure where submitted workloads are typically unknown and no a priori assumptions can be made about their behavior.

A different track to side-step DC workload unavailability is the design of open-source versions of popular applications, which resemble their behavior and structure. Examples of such workloads are Lucene [182] and Nutch [205] for Websearch, Roundcube [226] for Webmail, or Hadoop [134] for MapReduce [69]. In the same spirit, CloudSuite [106] is an open-source benchmark suite that aggregates a set of such applications, including data analytics, media streaming and web serving. While open-source applications cannot be exact replicas of production-class workloads, they provide a reasonable approximation of their behavior.

Interference-related workloads: Recent work has shown that reducing interference is critical to preserving application performance in DCs [76, 120, 187, 255]. Govindan et al. [120] designed a synthetic cache loader to profile an application’s

cache behavior and the pressure it would put on co-scheduled workloads. Similarly, to demonstrate the impact of interference in the memory subsystem, Mars et al. [187] designed two microkernels that create tunable contention in memory capacity and memory bandwidth. These kernels are then used to quantify the sensitivity of a workload to memory interference. Additionally, Tang et al. [255] designed Smash-Bench, a benchmark suite for cache and memory contention. Benchmarks include operations on binary search trees (BSTs), arrays and 3D arrays.

With iBench we extend the resources in which interference is quantified to the core, the memory hierarchy, and the storage and networking subsystems. This enables iBench to provide critical insights on the sensitivity of applications to resource contention that can guide both software (e.g., scheduling) and hardware (e.g., server provisioning) system studies.

5.3 iBench Workloads

5.3.1 Overview

The goal of iBench is to identify the shared resources an application creates contention to, and similarly the type and amount of contention the application is sensitive to. For this purpose, all iBench workloads have tunable intensity that progressively puts more pressure on a specific shared resource until the behavior of the application changes (i.e., performance degrades). A similar technique has been shown to provide accurate estimations on sensitivity to contention in the memory subsystem [187, 255]. In total, iBench consists of 15 carefully-crafted workloads, which we call sources of interference (SoIs), each for a different shared resource. The following section describes each one of them in detail. To provide some proportionality between the intensity of the benchmark and its impact on the corresponding resource, SoIs are designed such that their impact increases almost linearly with the intensity of the benchmark. Finally, we try to ensure that the impact of the different iBench workloads is not overlapping, e.g., that the memory bandwidth SoI does not cause significant contention in memory capacity and vice versa. Section 5.4 validates that this is indeed the case across SoIs.

5.3.2 Designing the SoIs

Memory capacity (SoI1): This kernel progressively accesses larger memory footprints until it takes over the entire memory capacity. The access pattern of addresses in this case is random, but can also be set to perform strided memory accesses. The following snippet shows the basic operation of SoI1:

```

t = 0;
while (t < duration) {
    ts = time(NULL);
    while (coverage < x%) {
        // SSA: to increase ILP
        access[0] += data[r] << 1;
        access[1] += data[r] << 1;
        ...
        access[30] += data[r] << 1;
        access[31] += data[r] << 1;
        wait(tx/accx);
    }
    x++;
    t += time(NULL) - ts;
}

```

The kernel identifies automatically the size of memory available in the system and scales its footprint “almost” proportionately with time. From the snippet above, t is the total time the SoI will run for. The benchmark uses single static assignment (SSA) to increase the ILP in memory accesses, and launches as many requests as necessary to guarantee the appropriate capacity coverage at each point during its execution, e.g., at 8% intensity, capacity coverage should be 8%. The memory addresses r are selected randomly with a low-overhead random generator function. For low intensities the kernel may switch to an idle state between memory requests. t_x is the time the kernel spends at a specific intensity level, and is a function of the benchmark duration

t and the intensity level x . acc_x is the number of accesses required to reach a specific coverage level and is also a function of the intensity x . The time the kernel can remain idle is proportional to t_x and inversely proportional to acc_x . As the kernel moves to higher intensities, the fraction of time the kernel remain idle reduces as more accesses are required to achieve a certain memory coverage. By default all kernels run for 10msec, however duration is a configurable parameter.

Memory bandwidth (SoI2): The benchmark in this case performs streaming (serial) memory accesses of increasing intensity to a small fraction of the address space. The intensity increases until the SoI consumes 100% of the sustained memory bandwidth of the specific machine. The intensity of accesses increases linearly with the memory bandwidth used. The reason why accesses happen to a relatively small fraction of memory (e.g., 10%) is to decouple the effects of contention in memory bandwidth from contention in memory capacity. The following snippet captures the main operation of the streaming kernel:

```
t = 0;
while (t < duration) {
    ts = time(NULL);
    for (int cnt = 0; cnt < acc_x; cnt++) {
        access[cnt] = data[cnt]*data[cnt+4];
        wait(t_x/acc_x);
    }
    x++;
    calculate acc_x;
    t += time(NULL) - ts;
}
```

The definition of t_x and acc_x is the same as before. In the subsequent SoIs we skip the code snippets in the interest of space, and describe their main operation.

Storage capacity (SoI3): Storage corresponds to the non-volatile secondary devices, e.g., disk drives or flash that store data. We assume these are disk drives for simplicity. The microbenchmark accesses random data segments across the disk's

sectors. The amount of accessed data increases linearly with the SoI's intensity, i.e., at 20% intensity close to 20% of disk capacity is accessed by the SoI.

Storage bandwidth (SoI4): This benchmark creates traffic of increasing intensity to the hard drives of the system. Disk accesses in this case are serial and the consumed disk bandwidth increases almost linearly with the intensity of the SoI, e.g., at 100% intensity, the SoI uses close to 100% of the sustained disk bandwidth of the system.

Network bandwidth (SoI5): This SoI is of interest to workloads with network connectivity, e.g., online services or distributed frameworks like MapReduce. It operates by issuing network requests of increasing intensity (size and frequency of requests) to a remote host. We currently do not deploy rate limiting mechanisms, therefore the SoI can take over 100% of the available network bandwidth, essentially starving any co-scheduled application.

Last level cache (LLC) capacity (SoI6): The benchmark mines the `/proc/cpuinfo` of the system and adjusts its footprint, access pattern and the pace that its intensity increases based on the size and associativity of the specific LLC. The kernel issues random accesses that cover an increasing size of the LLC capacity. Because caches are structured in sets, it is easy to mathematically prove and practically guarantee that the footprint of the benchmark increases linearly with the intensity of the SoI and that its accesses are uniformly distributed. We skip the proof in the interest of space. Finally, to guarantee that accesses are not intercepted in the lower levels of the hierarchy (L1, L2) we concurrently run small tests that sweep the smaller caches (without introducing additional misses) to ensure that all accesses from the SoI go to the LLC.

LLC bandwidth (SoI7): This benchmark is similar to the SoI for memory bandwidth in that it performs streaming data accesses to the LLC. In this case the size and peak bandwidth the SoI targets are tuned to the parameters of the specific last level cache. Because accesses are streaming over a fraction of the cache, the lower levels of the hierarchy do not play as important a role as with random accesses. We have found that running the sweep tests for L1 and L2 does not make a significant difference when measuring sensitivity to contention in LLC bandwidth.

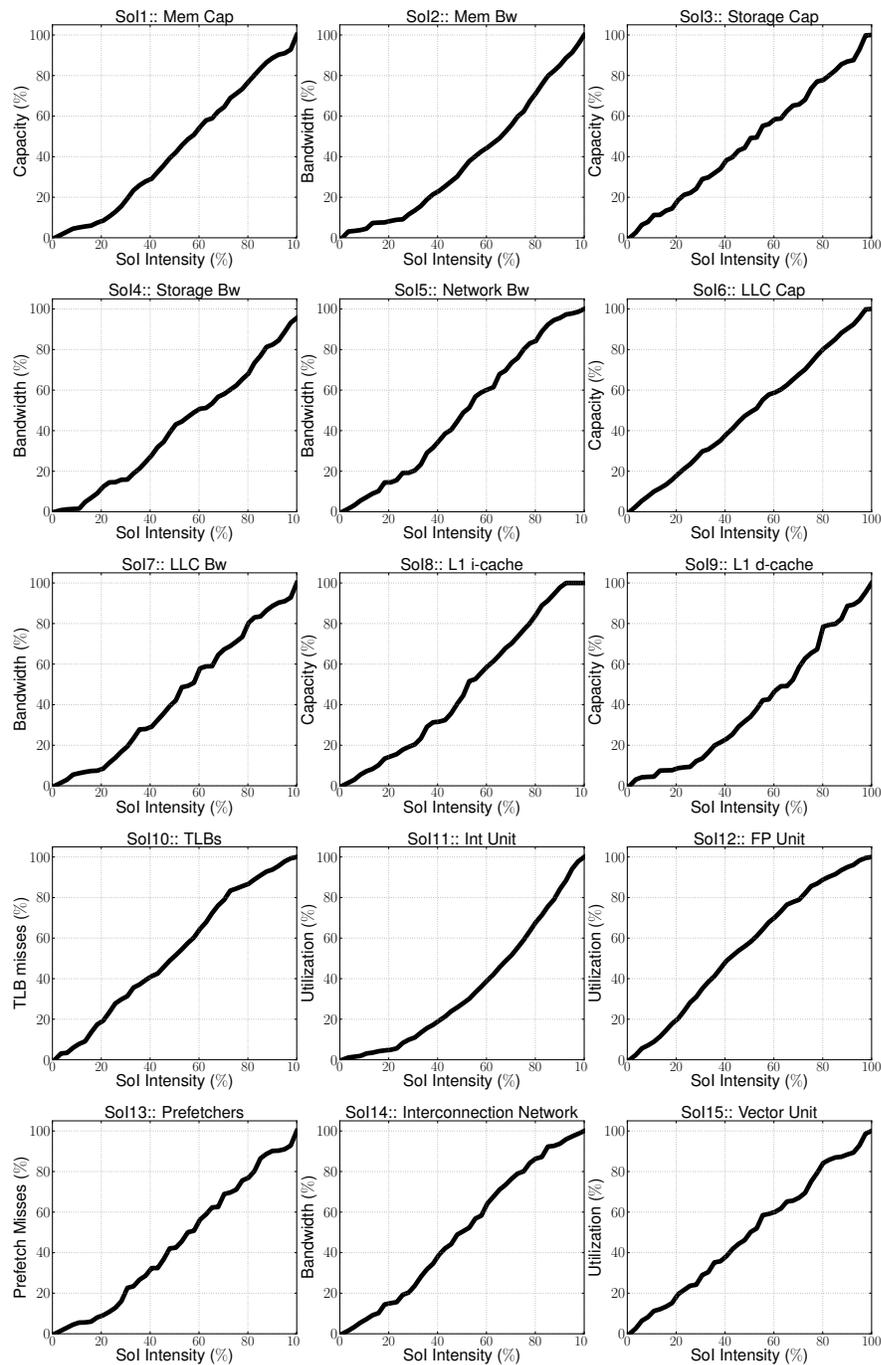


Figure 5.2: The iBench workloads. For each benchmark we show the system impact for increasing Sol intensity. We do not include the graphs for the L2 capacity and bandwidth SolIs. These are similar to the ones for LLC capacity and bandwidth.

L2 capacity (SoI6’): This is a similar benchmark to SoI6 (LLC Capacity), and is applicable in systems with 3+ levels of cache hierarchy. The footprint in this case grows up to the L2 cache size and the L2 associativity is used to tune how intensity changes over the kernel’s duration.

L2 bandwidth (SoI7’): Similar to SoI7 (LLC bandwidth), but tuned to the size and associativity of the L2. Accesses in this case are streaming.

L1 i-cache (SoI8): A simple kernel that sweeps through increasing fractions of the i-cache, until it populates its full capacity. Accesses in this case are again random.

L1 d-cache (SoI9): A copy of the previous SoI, tuned to the specific structure and size of the d-cache (typically the same as the i-cache).

Translation lookahead buffer (TLB) (SoI10): This benchmark fetches pages from memory at increasing rates until it occupies all the TLB entries. This forces long page walks for any co-scheduled application, inducing high performance degradations. Again, because of the structure of TLBs it is easy to compute the pace at which SoI intensity should increase to guarantee a linear relation with the occupied entries.

Integer processing units (SoI11): While the core can be approached as a single shared resource, we decide to separate the different types of operations to integer, floating point (and an optional vector SoI when SSE extensions are available). All three SoIs are assembly-level benchmarks that issue an increasing number of the corresponding type of instructions. For SoI11 these are instructions between integers.

FP processing units (SoI12): Similarly here, floating point instructions are issued at an increasing rate. SoIs 11 and 12 (and 15 when applicable) can run both on the same hardware thread and on different threads sharing the same core.

Prefetchers (SoI13): This benchmark tries to inject unpredictability in the instructions the prefetcher brings from memory, and decrease its effectiveness. This may seem similar to the operation of the L1 i-cache benchmark, however the prefetcher SoI employs a different access pattern than SoI8. Instead of simply sweeping through the L1 and evicting the co-runner’s instructions, the SoI here is a small program that only takes up a fraction of the L1 i-cache, but interleaves its instructions with the examined application’s instructions. This way the prefetcher gets tricked into

bringing the SoI’s next “expected” instructions from memory instead of the primary application’s. Intensity here translates to the time the SoI is non-idle. This SoI also interacts in part with the system’s branch predictor.

Interconnection network (SoI14): This benchmark is designed using message passing primitives between cores. As the SoI intensity goes up the number and fanout of messages sent by the kernel increases. For high intensities the injected traffic becomes adversarial, leading the remaining system cores to starvation.

Vector processing units (SoI15): This SoI is only applicable in systems with SIMD ISA extensions, e.g., SSE3/4. It takes advantage of these extensions to launch 256-wide SIMD instructions with increasing frequency. Instructions are issued on a small data set to avoid interfering with the cache/memory hierarchy. None of the systems we tested uses extensive memoization techniques therefore operating on the same data does not reduce the load to the vector units.

5.4 Validation

We want to validate three aspects of iBench; first that the benchmarks indeed induce contention in their corresponding resources, and that their impact increases almost linearly with their intensity. Second, we want to evaluate the impact of iBench on conventional and DC applications and verify that the SoIs can be used to detect sensitivity to interference. Finally, we want to verify that the different SoIs do not overlap with each other in a way that voids the insights drawn about an application’s behavior, e.g., that the memory bandwidth SoI does not introduce significant contention in memory capacity.

5.4.1 Individual SoIs Validation

Figure 5.2 shows the impact of the 15 SoIs across their intensity spectrum (0-100%). For capacity-related benchmarks we show their cache, memory or disk footprint. For the bandwidth-related SoIs we show the fraction of bandwidth they consume normalized to the provisioned sustained cache, memory or disk bandwidth. For the

core-related SoIs we show the utilization they induce in the corresponding functional units (int, fp or vector). Finally, for the TLB benchmark we show TLB misses and for the prefetcher benchmark, prefetch misses. All measurements are collected using performance counters on a dual-socket, 8-core Nehalem server with private L1s and L2s and a shared 8MB L3 cache and 32GB of RAM. The server has a 1GB NIC and 4 500GB hard drives. Each SoI runs for 10msec on its own and covers its full range of 0 to 100% of intensity. Each SoI automatically detects the system parameters that it needs in order to adjust its operation, e.g., cache or TLB size, core count or NIC type. From Figure 5.2 we see that for all benchmarks the impact to the corresponding resource increases almost linearly with their intensity. The only SoIs that slightly deviate from linear are the core-related benchmarks Int and FP. This happens because correlating the number of issued instructions to the eventual system utilization is harder than correlating the number of cache accesses to the capacity used. We plan to further refine these workloads to better approach linear load increase as part of future work.

5.4.2 SoI Impact on Applications

iBench is aimed to detect and quantify the sensitivity of DC and conventional workloads to various sources of interference. Here we validate that this operation is accurate. We inject iBench workloads in a conventional application (mcf from the SPEC CPU2006 suite) and in a DC latency-critical application (memcached [107]) and measure their sensitivity to interference in the corresponding resources. Each application runs on a single server, and memcached is set up with 1000 clients launching 40,000 QPS in total, with a target per-request latency of 200usec. mcf is profiled for 10msec against the LLC capacity SoI, and memcached against the network bandwidth SoI. Both SoIs inflate to full intensity (100%). Figure 5.3a, b shows the results for mcf and Figure 5.3c, d for memcached. Figure 5.3a shows the performance impact of contention in LLC capacity for mcf and the corresponding miss rate curve as the intensity of the SoI increases. Comparing the two shows that the SoI indeed induces significant performance degradation to the application due to cache contention. The

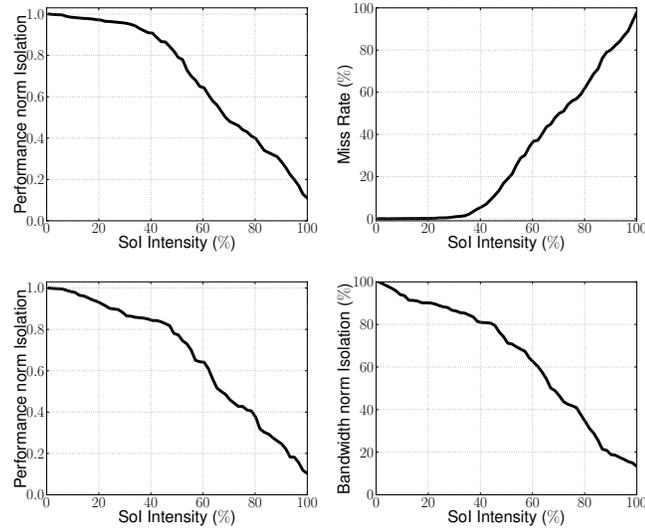


Figure 5.3: Validation of the impact contention generated using iBench has on mcf and memcached. Figure 5.3a shows the performance of mcf when co-scheduled with the LLC capacity SoI, while Figure 5.3b shows its new miss rate curve as SoI intensity increases. Figure 5.3c shows the performance of memcached when running with the network bandwidth SoI and Figure 5.3d shows its bandwidth share compared to when running alone.

point when performance gets a significant hit coincides with the moment when the miss rate increases rapidly, therefore the SoI is correctly stressing its target resource. Similarly, for memcached we show the performance impact from increased contention in the network and the bandwidth fraction memcached manages to extract compared to the target fraction it needs to preserve its performance requirements. Again there is a direct correlation between performance degradation and its cause. As SoI intensity increases, the goodput of memcached (fraction of requests that meet their target latency) rapidly decreases. Figure 5.3d shows the reason behind this degradation. For high SoI intensities, the bandwidth share of memcached becomes increasingly smaller, introducing queueing delays to incoming requests. At the same time, examining its cache miss rate or memory behavior does not show significant variations compared to when memcached is running alone. This verifies that the SoI is confined to its specific resource and does not violently disrupt the utilization of other subsystems. We further validate this observation in the following subsection. We have also verified

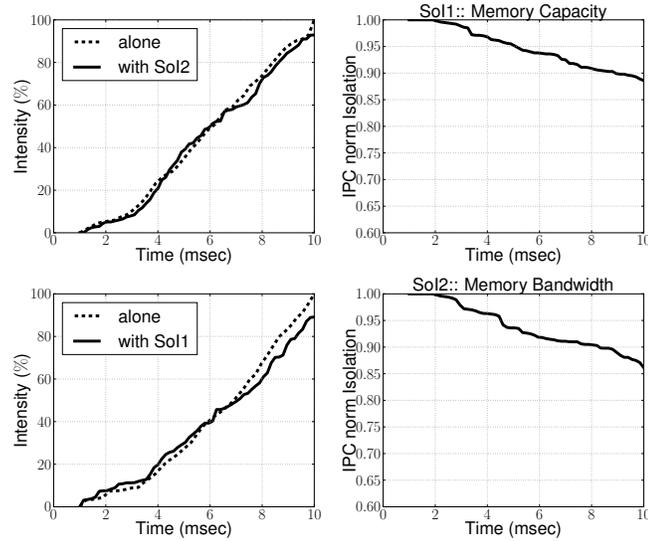


Figure 5.4: Validation of the impact SoIs have on each other. Figure 5.4a shows the intensity of So1 when co-scheduled with So2 and similarly for So2 (Figure 5.4c). Figure 5.4b, d show the achieved IPC normalized to target when the two SoIs run together. Overall, interference between benchmarks is minimal.

that these results are consistent across the different SoIs for various workload types.

5.4.3 Correlation between SoIs

Finally, we verify that different sources of interference (SoIs) do not overlap and interfere with each other. For this purpose we co-schedule two SoIs at a time in the same core of the 8-core system previously used. Figure 5.4 shows the increase in intensity and corresponding performance normalized to isolation for a co-schedule of the memory capacity and memory bandwidth SoIs. As shown in the figure, for high system loads, there is a small impact in the ability of each SoI to reach its full intensity. Similarly there is a slight degradation in performance compared to running in isolation. However, for both SoIs degradations are mild, which means that the different benchmarks do not induce significant contention outside their target resource. This is important to both obtain accurate interference measurements, and make valid assumptions on their causes. We have performed this experiment with different SoI combinations with similar results.

5.5 Use Cases

5.5.1 Datacenter Scheduling

Currently, DC operators often disallow application co-scheduling in shared servers to preserve QoS guarantees. However, this leads to serious resource underutilization. On the other hand, co-scheduling applications can induce interference due to contention in shared resources. We use iBench to quantify the tolerance a workload has to various sources of interference, and similarly the interference it causes in shared resources. Given this information, a scheduler determines the applications that can be safely co-scheduled without performance degradation from interference. For this use case, the scheduler simply tries to minimize:

$$\|i_t - i_c\|_{L_1} \tag{5.1}$$

where i_t and i_c the tolerated and caused interference for two examined applications. The tolerated interference is calculated as described in Section 5.4. The caused interference is similarly calculated, by quantifying the impact the examined workload has on the performance of an SoI. The L_1 norm is calculated across the different SoIs. More sophisticated scheduling techniques can be deployed to take better advantage of the information provided by iBench [76]. Obviously applications can change behavior during their execution. This is especially true for DC workloads [31, 191]. The scheduler adapts to these changes to preserve QoS throughout an application's execution. If at any point in time it detects that an application is running under its QoS, the scheduler injects iBench workloads to the system to construct a new interference profile. Any further scheduling decisions use the new interference profile. Required migrations due to behavior changes are handled by a low-overhead live migration system present in the cluster. In the event where migration is not possible, the scheduler disallows additional applications to be placed on the same machine as the affected workload.

We design three scenarios; first a cloud workload mix that resembles a system like EC2, where 200 applications are submitted in a 40-machine cluster with 1 sec

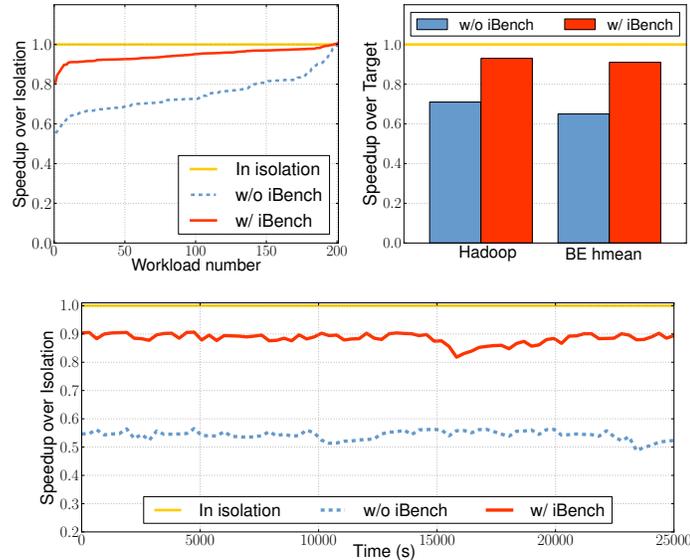


Figure 5.5: Performance achieved by a scheduler that accounts for interference using iBench compared to a system that ignores interference in scheduling decisions. Results are shown for three scenarios: a cloud workload mix (Figure 5.5a), a distributed workload (Figure 5.5b) and a latency-critical application (Figure 5.5c).

inter-arrival times. All nodes are dual socket, 4-12 core machines with private L1 and L2 caches and shared L3 caches, and 16-48GB of RAM. All applications are selected randomly from a pool consisting of the full SPECCPU2006 suite, 22 workloads from PARSEC [40], SPLASH-2 [282], BioParallel [150] and Minebench [198], 140 multiprogrammed workloads of 4 SPEC applications each, based on the methodology in [228], and 10 I/O-bound data mining workloads [219]. The second scenario involves a Hadoop workload running distributed on 40 nodes with low-priority best-effort (BE) applications occupying the remaining server capacity, and the third scenario involves a 40-node installation of *memcached*, running as the primary process and best-effort applications using the remaining resources. Figure 5.5a compares application performance for the first scenario when quantifying interference using iBench, against a baseline scheduler that only considers the CPU and memory requirements of an application and assigns workloads to least-loaded (LL) servers (*w/o iBench*). The latter is common practice in many cloud providers today [271]. Performance is normalized to running in isolation and applications are ordered from worst- to best-performing.

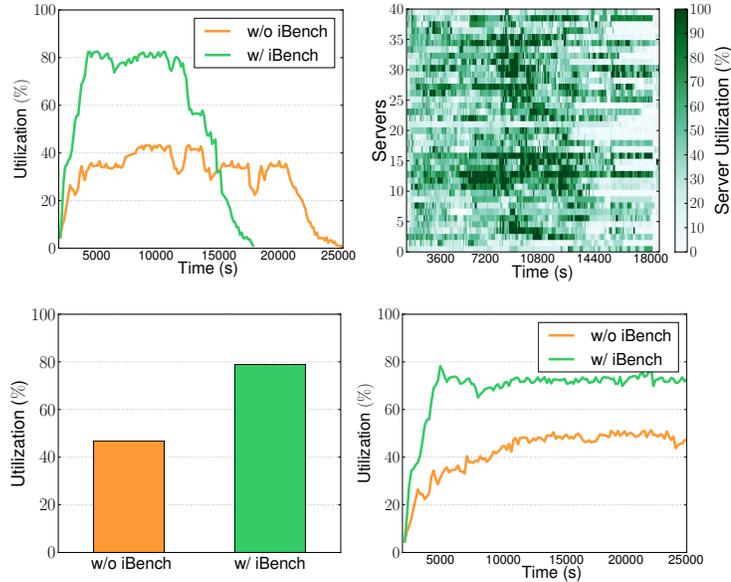


Figure 5.6: Utilization achieved by a scheduler that uses iBench compared to a system that ignores interference in scheduling decisions. For the first scenario fewer machines are needed (and for less time) (Figure 5.6a, b), while for the second (Figure 5.6c) and third scenarios (Figure 5.6d) more best-effort applications are co-scheduled with the primary workload.

Using iBench to quantify the pressure applications put on various system resources improves performance, by 15.7% on average and up to 25%. Similarly, performance improves in the second and third scenario both for the primary workloads (Hadoop and memcached respectively) and the best-effort applications. Managing interference is beneficial to utilization as well, since more applications can be scheduled on the same machine. Figure 5.6 shows the utilization for each of the three scenarios when accounting for interference using iBench and when using the baseline least-loaded (LL) scheduler. The benefits are twofold; first utilization increases, improving resource-efficiency for the DC operator (Figure 5.6a). Second, the duration of the scenario reduces because applications are running near their target performance. Figure 5.6b offers a closer look at utilization across the different servers in the cluster throughout the scenario’s execution. The increase in utilization is also consistent for the other two scenarios (35.6% and 27.1% respectively on average). There is still some performance degradation in these scenarios, which iBench cannot prevent. This is due to

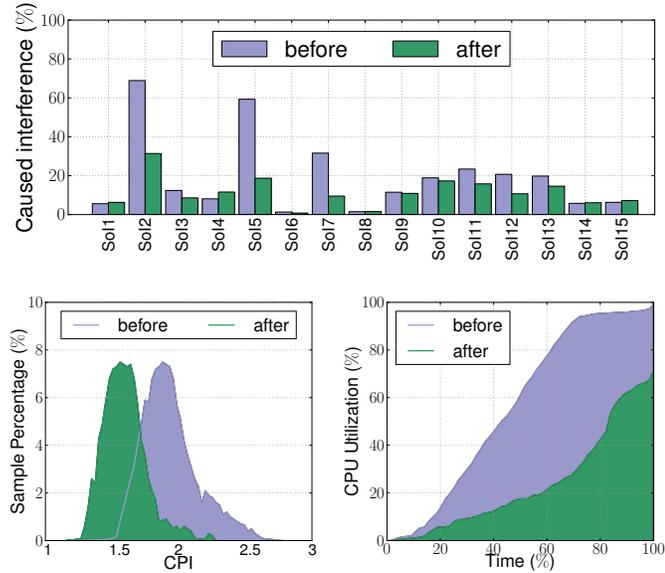


Figure 5.7: Using iBench to provision a server that hosts a specific workload improves performance and reduces resource contention. Figure 5.7a compares the old and new interference profiles of the workload. Figure 5.7b shows the CPI distribution for memcached in the original system configuration and after reconfiguring the system based on the interference profile from iBench, and Figure 5.7c shows that utilization decreases, as resources are appropriately balanced to reduce contention.

fast-changing workloads, complex applications that introduce inaccuracies in interference measurements, or workloads that have pathologies when co-scheduled with specific applications. Additional mechanisms can be used to address these issues.

5.5.2 Server Provisioning

Provisioning servers is especially difficult for cloud providers that have to accommodate any - possibly unknown - submitted workload. Even in the case of well-studied, long-running applications the datacenter architect must deal with evolving application code and varying user patterns. Here we use the output of iBench to guide the way system resources are balanced in a DC server running memcached [107]. The workload runs with 1000 clients launching a total of 40,000QPS with a latency constraint of 200usec. Figure 5.7a shows the interference profile of the application running on a default server configuration (4 cores, 8MB L3, 16GB RAM, 1GB NIC) across the

different SoIs. It is evident that the application puts significant pressure on the cache hierarchy and the network and memory subsystems (SoI2: memory bandwidth, SoI5: network bandwidth, SoI7: LLC bandwidth and SoI11-12: core). Based on this information we adjust the parameters of the system. To alleviate the contention in the memory hierarchy we switch to a triple-memory channel server with 24GB of total memory capacity. Similarly, we move from a server with a 1GB to a 10GB NIC to accommodate the application's network demands. We maintain the core count and the rest of the system parameters the same. Figure 5.7a also shows the new interference profile, where both the contention in the cache/memory hierarchy and the network subsystem are now significantly reduced. Figure 5.7b shows the distribution of CPI in the default server configuration and in the server provisioned based on the output of iBench, while Figure 5.7c compares the CPU utilization in the two systems. In both cases accounting for contention when provisioning the system improves application performance (the CPI curve is shifted to the left in the new system) and reduces CPU throttling due to memory stalls. Similarly, we can use the information on resource contention to guide the microarchitecture design (cache hierarchy, pipeline organization, etc.) of hardware aimed to service a particular application.

5.5.3 Application Development/Testing

An important reason behind resource inefficiency is poor application design. Workloads are often written without sufficient considerations of sensible resource usage, resulting in unnecessarily bloated code, huge memory footprints, and high CPU utilization. This problem is even more prominent in DC workloads, which are often complex, multi-tier applications with several interdependent components. Despite the long testing periods devoted to these workloads, robustness and performance are typically the main optimization objectives, with resource-efficiency being less important. Here we show that using iBench to identify code regions that cause high contention not only improves efficiency by eliminating unnecessary resource consumption, but is also beneficial to performance by reducing resource contention.

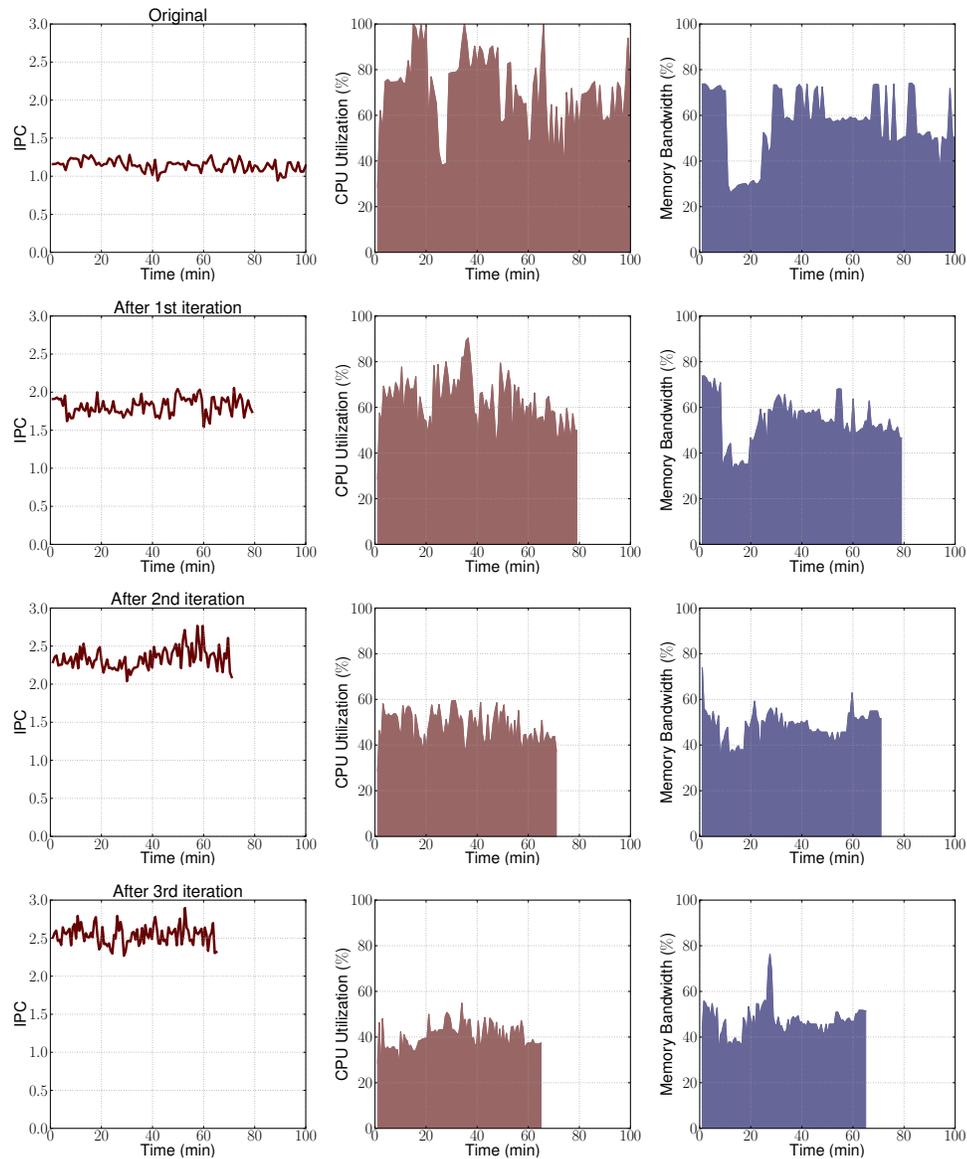


Figure 5.8: Performance (IPC), CPU utilization and memory bandwidth utilization for the testing application across three optimization iterations using iBench. While performance for the original application is low, with the CPU being saturated, identifying contentious regions in the code progressively improves throughput and decreases resource utilization, improving resource efficiency.

For this purpose we start with an unoptimized data mining application that performs collaborative filtering on a large dataset of sparse data. The data are movie ratings from 180k users. Running the original version of the code, which relies on Singular Value Decomposition and PQ-reconstruction [219, 43] results in very high contention in LLC capacity, bandwidth, L1 d-cache and L2 cache capacity and bandwidth, memory bandwidth and FP computation. Running the program to completion takes approximately 1.6h. The performance of the original code is shown in Figure 5.8 (first row, leftmost figure). The second and third figures in the first row show the CPU and memory bandwidth utilization of the program (normalized to sustained memory bandwidth for the server). After detecting the points of contention using iBench, we optimize parts of the code to make better use of system resources. In the first code iteration we switch to SIMD operations using SSE4 [201]. As shown in the second row in Figure 5.8 both performance and resource efficiency benefit. The boost in performance comes from leveraging spatial locality in matrix accesses, while the decrease in required resources comes from performing fewer operations on larger chunks of data and reducing the misses to the cache hierarchy. We now repeat the interference characterization for the new program. iBench again helps identify remaining inefficiencies in the code that induce resource contention. We progressively address these with optimizations such as reordering of operations to the matrix elements or memoizing intermediate results. After each iteration we reevaluate the application’s performance and resource utilization. As shown in the last row of Figure 5.8 the final code runs in 35% less time than the original unoptimized version *while* requiring fewer system resources. While the code optimizations shown here are relatively straightforward, we believe that given the speed of obtaining the interference profile, using signals from iBench can significantly facilitate the development and testing process of large applications.

5.5.4 Scheduling in Heterogeneous CMPs

Finally, we show that iBench is applicable outside DC system studies. CMPs today consist of tens of - often - heterogeneous cores [263, 239, 280, 301]. Scheduling for these

systems is challenging because in addition to the interference between applications that share resources, the scheduler should account for system heterogeneity. Similarly to the first use case, we design a simple scheduler that takes the interference profile obtained by iBench and identifies how each application from a multiprogrammed mix should be mapped to heterogeneous cores. When the mix is submitted to the system, each workload is briefly profiled against the iBench workloads to obtain its interference profile. Each SoI requires at most 10msec and runs can be done in parallel by replicating and sandboxing the application binary. Profiling can additionally leverage classification techniques to reduce the training overhead, by only profiling against a subset of SoIs and deriving the missing entries based on similarities with previous applications [34, 76]. We first create 40 4-SPECCPU2006 application mixes and schedule them on a simulated 4-core CMP [228] with 2 Xeon-like and 2 Atom-like cores from different generations each. The simulator captures contention in the cache and memory hierarchy, therefore the same process as before is used to quantify the impact of interference on application performance. The examined workloads do not exhibit storage or network activity hence we do not use the SoIs creating contention in those resources (SoI3-5). SPEC workloads are classified with regards to their cache demands as insensitive (n), friendly (f), fitting (t) and streaming (s), and mixes are created based on the methodology in [228]. Cores differ in their frequency, private cache hierarchy and microarchitectural details (e.g., pipeline, prefetchers, branch predictors, issue width). All cores share an 8MB last level cache (LLC) and 16GB of memory. The scheduler uses iBench to identify the type of core and co-scheduled applications that constrain interference and selects the mapping that minimizes the average interference across workloads. Although this is not necessarily a global optimum it is good enough that performance does not degrade and utilization increases. The scheduler can also take advantage of workload signatures [263] to further refine the application-to-core mapping search space. We also create 60 16-application mixes and schedule them in a similar system with 16 cores. The variability in frequencies and cache hierarchies here is more widespread. Figure 5.9 shows the performance obtained when using iBench to guide the scheduling decisions. The upper figures show the performance of the 4-app mixes ordered from worst to best-performing compared

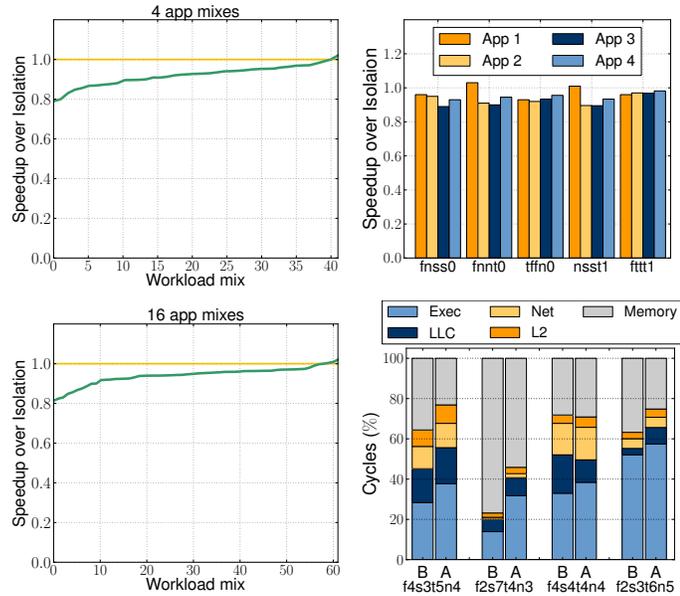


Figure 5.9: Scheduling in heterogeneous CMPs. The upper figures show performance across the 4-application mixes and a per-application breakdown for selected mixes. The lower figures show the performance for the 16-application mixes and a breakdown of execution time to various subsystems for select mixes, before (B) and after (A) the use of iBench for scheduling.

to isolated runs, and the breakdown to per-application performance for selected mixes. Performance degradations are marginal for most workloads. The lower figures show the performance across the 16-app mixes and the breakdown of clock cycles to the various subsystems for selected mixes. While without the use of iBench several mixes spend significant fractions waiting in memory instead of executing instructions, by minimizing interference larger fractions of time are devoted to useful execution. We plan to perform a more detailed study of scheduling tradeoffs in heterogeneous CMPs as part of future work.

5.6 Conclusions

We presented iBench, a benchmark suite that measures the tolerated and caused interference of a workload in various shared resources. iBench is geared towards DC applications, but can also be applied to conventional workloads. It consists of 15

benchmarks (SoIs) that induce pressure over a wide range of shared resources that span the core, cache hierarchy, memory, storage and networking subsystems. iBench quantifies the type and degree of interference that an application generates in this set of shared resources. Similarly, it measures the type and intensity of interference an application can tolerate before violating its QoS across the same resources. We have validated the accuracy and consistency of iBench against a number of DC applications, ranging from conventional single-node applications, to distributed Hadoop workloads, and latency-critical online services. We have also evaluated a number of use cases for iBench. First, we use the interference information obtained with iBench to schedule workloads in an EC2-like environment in a way that minimizes interference between co-scheduled applications and improves system utilization. Second, we have shown how iBench can assist towards making informed decisions on the hardware specifications of a chip aimed for DC workloads, or on the provisioning of a DC server. Third, we have shown how iBench can be used by software developers to design more resource-efficient applications during testing. Finally, we have shown that iBench is applicable outside the context of DCs, and have used it for scheduling in large-scale heterogeneous CMPs. In all cases, using iBench significantly improves the decision quality, performance, and resource efficiency of the system.

Chapter 6

ARQ: QoS-Aware Admission Control

6.1 Introduction

An increasing amount of computing is performed in the cloud, primarily due to cost benefits for both the end-users and the operators of datacenters (DC) that host cloud services [31]. The operator of a cloud service must schedule the stream of incoming applications on available servers in a *resource-efficient* manner, i.e., achieving fast execution (user’s goal) at high resource utilization (operator’s goal). This scheduling problem is particularly difficult for several reasons, including diverse application characteristics [31, 163], insufficient workload knowledge, co-scheduled application interference and platform heterogeneity. An additional challenge occurs during periods of adversarial traffic, i.e., intervals with very high load, when the system can become oversubscribed, resulting in poor performance. Most DCs employ some admission control to minimize such effects.

DC users are interested in two performance metrics; how fast the application starts running (*waiting time*) and how fast it completes thereafter (*execution time*). While recent work has shown how to improve execution time in the presence of unknown workloads, varying interference sensitivities and heterogeneous servers [76], it does not solve the “head of line blocking” problem [232]. Additionally, some applications have

strict scheduling deadlines, while others can tolerate delays in order to be assigned to preferred servers. In all cases, resource requirements should be taken into account at admission point [47].

We propose *ARQ* (*Admission control with Resource Quality-awareness*), a QoS-aware admission control protocol that builds on Paragon and accounts for the resource quality an application needs to preserve its QoS. Resource quality reflects the additional load a server can support without violating application QoS, given its configuration and the applications it currently hosts. For example, a server hosting a low-latency key-value store and a relational database has high resource quality for a Spark job, if it can accommodate it without any performance violations for the new or previous applications. ARQ divides workloads into multiple classes and directs them to different queues. This way demanding workloads do not block easy-to-satisfy applications, as they wait for an appropriate server to become available. On the other hand, since DC applications have strict QoS guarantees, they can only be queued for limited amounts of time, while waiting for an appropriate server. ARQ detects when an application is about to violate its performance requirements and re-directs it to a different queue before the QoS violation occurs. We explore the trade-off between waiting time and quality of resources and solve the corresponding optimization problem to find the optimal switching point.

We evaluate ARQ both in small and large-scale experiments. First, we compare the system without and with ARQ in a local cluster with 40 machines and show the benefits in performance and efficiency. We also evaluate ARQ on a 1000-server cluster on Amazon EC2. For an oversubscribed scenario with 8500 applications, Paragon with ARQ guarantees that 99% of workloads have less than 10% performance degradation, while improving utilization by 46%.

6.2 Background

As we described in Chapter 3, Paragon is a QoS-aware scheduler that accounts for server platform heterogeneity and interference between co-scheduled applications.

While accounting for these factors allows Paragon to improve application performance, and cluster utilization, the scheduler has no logic to decide when applications should be admitted and scheduled. Paragon accounts for workload characteristics to decide where to assign a workload, but it does not solve the “head of line blocking” problem that can cause high waiting times. By default, applications are scheduled in a simple FIFO order. This has two shortcomings; first, easy-to-satisfy workloads can get trapped behind demanding applications, e.g., workloads that require exclusive instances of high-end, multi-socket servers to preserve their QoS. Second, in the event of an oversubscribed scenario, i.e., when the required resources are more than the total resources available in the system, Paragon implements an application-agnostic admission control protocol. It queues applications in a single queue until the first server becomes available, and then resumes FIFO-ordered scheduling. This ignores the fact that applications need resources of a certain quality to meet their QoS, and can result in performance degradation.

6.3 Admission Control

6.3.1 Overview

Large cloud providers such as Amazon EC2 and Windows Azure, typically deploy some admission control protocol. This prevents machine oversubscription, i.e., the same core servicing more than one application, resulting in high interference and QoS violations.

We design ARQ, a *QoS-aware admission control protocol* that queues and schedules applications based on the quality of resources they need. This solves two problems; first, applications that demand few, easy-to-satisfy resources are not blocked behind demanding workloads. Second, if no suitable servers are available for a given application, the workload waits for a server of appropriate quality to be freed. Alternatively, the application would be directed to the first free server to avoid queueing delays, with the risk of performance losses.

Resource quality: The resource demands of a workload reflect the load a server

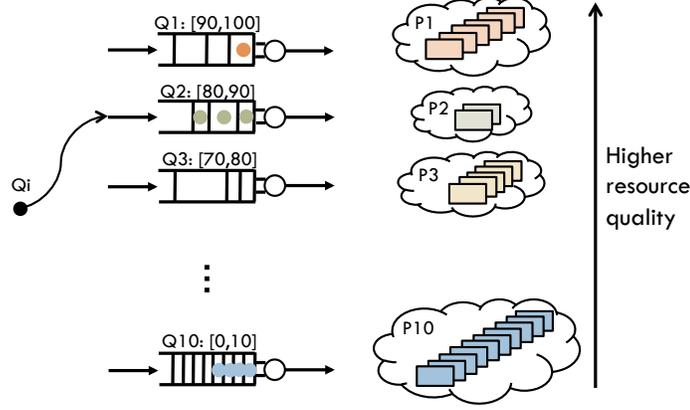


Figure 6.1: ARQ design. Each queue corresponds to applications with different resource quality requirements.

should support for the application to meet its QoS. This is a function of the interference the server can tolerate from the new application, and the interference the new workload can tolerate from applications already running on the machine. We use the classification engine of Paragon to derive the interference each server tolerates (t_k) and the interference each application causes (c_k) on a set of shared resources. Shared resources include the cache and memory hierarchy, CPU modules and storage and network devices. Details on how c_k 's and t_k 's are obtained can be found in Chapter 3. The interference profile of a server is updated upon initiation or completion of an application's execution. This information guides scheduling decisions by assigning applications to suitable servers. Given the interference profile of application i , we define *resource quality* as:

$$Q_i = \sum_k c_k \quad (6.1)$$

Similarly, resource quality for a server is defined as the sum of t_k over the different shared resources. Q_i s are normalized in 0 to 100%. Conceptually, high Q_i reflects applications sensitive to interference, that need high quality resources. Low Q_i on the other hand, corresponds to workloads that are insensitive to interference, and can satisfy their QoS even when assigned to servers of poor resource quality, e.g.,

highly-loaded machines, or machines with few cores.

Multi-class admission control: We design ARQ as an admission control protocol with multiple classes of “customers” [22, 39, 138, 164, 192], where customers in this case correspond to applications. The class an application belongs to is determined by its Q_i value. Applications with Q_i values that fall in the same range are assigned to the same class. We assume ten classes of applications for now, and justify this selection in the evaluation section (see sensitivity study in Section 6.5). Figure 6.1 shows an overview of ARQ. Each queue corresponds to applications of a specific class. From top to bottom we move from more to less demanding applications. Upon arrival, the cluster manager determines the class an application belongs to and queues it appropriately. Each class has a corresponding server pool of appropriate resource quality. Separating applications based on their resource quality requirements helps ARQ resolve bottlenecks where applications that are sensitive to interference block workloads that are not. On the other hand, applications cannot be queued indefinitely waiting for the perfect server. We address this issue by diverting workloads to queues with better or worse resource qualities.

6.3.2 Waiting Time versus Resource Quality

Diverging an application to a different queue creates a trade-off between the time an application is waiting in a queue, and the quality of resources it is allocated. We approach this trade-off as an optimization problem.

Queue bypassing: When there is no available server in the pool of a class, queued workloads should be diverted to another queue. There are two possible options for where a workload can be redirected. First, it can be *diverted to a higher queue*. If the queue directly above the queue the workload was originally placed in is empty, the workload is assigned to one of its servers. This hurts utilization, since resources of higher quality than necessary are allocated, but preserves the workload’s QoS requirements. In the opposite case the workload is *diverted to a lower queue*. In that case, performance may be degraded, since the application receives resources of lower quality than required. However, the scheme guarantees that in all cases the

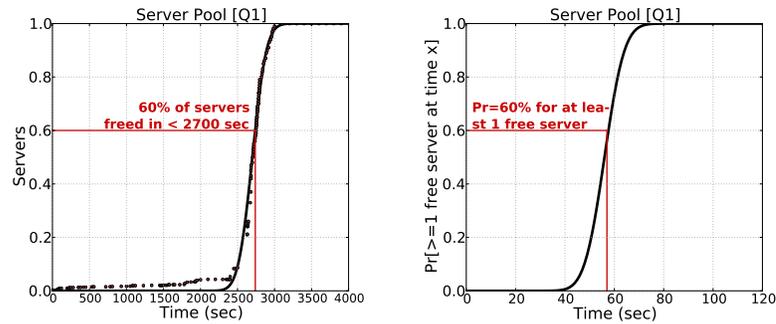


Figure 6.2: CDF of server busy times and CDF of the probability that there will be at least one free server within a specific time window from an application’s arrival.

application will be assigned to a server within the time window dictated by its QoS constraints.

Free-server probability distributions: ARQ needs to know the likelihood that a server of a specific class will become available within the time an application can be queued for, to decide when the workload should be diverted to the next queue. We statistically analyze the server busy time periods for each server pool to obtain these probability distributions. Busy periods are defined as the per-server time intervals from the moment a server is assigned a workload, until that workload completes.

We first use *distribution fitting* to represent the per-pool server busy time in a closed form using known distributions. Figure 6.2a shows the CDF of server busy time for the first server pool (highest quality servers) in a 1,000 server experiment. More details on the methodology can be found in Section 6.4. We show the experimental data (dots) and the closed form representation, derived from distribution fitting. In this case, the data is fitted to a curve resembling a normal distribution. The CDF reflects the fraction of servers that are freed within some time after they have been allocated to an application. For example, 60% of servers in this server pool are freed within 2700 sec from the time an application is scheduled to them.

Using this closed form CDF we easily derive the free-server CDF, which reflects the *probability that within a time interval from an application’s arrival, at least one server of the corresponding pool will be available*. Figure 6.2b shows the free-server probability CDF for the first server pool. The highlighted point shows that there is a

60% probability that within 56 sec from an application’s arrival to that queue, there will be at least one free server in the pool. Free-server CDFs are updated during workload execution to capture changes in application behavior.

Switching between queues: ARQ determines the switching point between queues with the objective to maximize the probability that a server becomes available within a certain window from an application’s arrival. For simplicity of explanation we assume that an application’s QoS is defined at $0.95x$ of the application’s optimal performance. This means that the workload can tolerate at most a 5% performance degradation. Scheduling deadlines or queries-per-second (QPS) can also serve as queueing constraints. Given the free-server CDFs for each server pool, ARQ solves the following optimization problem for application a , switching between queues i and j :

$$\begin{aligned} \max \{ & (S_a - wt_i(t)) \cdot Q_i \cdot Pr_i[t], (S_a - wt_j(t)) \cdot Q_j \cdot Pr_j[t] \} \\ \text{s.t. } & (wt_i(t) + wt_j(t) + P_a) < 0.05 \cdot CT_a \end{aligned}$$

where $Pr_i[t]$ is the probability that there is a free server in queue i , Q_i is the resource quality of queue i , CT_a is the optimal execution time for application a , P_a is the classification overhead of Paragon, and $S_a = 1.05 \cdot CT_a - P_a$ is the available “slack” that can be used for queueing, before the application violates its QoS constraints. ARQ finds the switching time that maximizes the probability that a server of either queue i or j will become available such that the application preserves its QoS guarantees. It also promotes waiting longer for a server of the same class rather than eagerly switching to the next queue ($Q_i > Q_j$).

In our analysis we assume batch, single-node applications. In the case of interactive or transactional workloads additional care must be taken to accommodate load changes, e.g., through VM migration. The scheduler detects such changes and adjusts workload placement to preserve QoS. Detection is based on SoI injection and application reclassification.

Server Type	GHz	sockets	cores	L1(KB)	LLC(MB)	mem(GB)	#
Xeon L5609	1.87	2	8	32/32	12	24 DDR3	1
Xeon X5650	2.67	2	12	32/32	12	24 DDR3	2
Xeon X5670	2.93	2	12	32/32	12	48 DDR3	2
Xeon L5640	2.27	2	12	32/32	12	48 DDR3	1
Xeon MP	3.16	4	4	16/16	1	8 DDR2	5
Xeon E5345	2.33	1	4	32/32	8	32 FB-DIMM	8
Xeon E5335	2.00	1	4	32/32	8	16 FB-DIMM	8
Opteron 240	1.80	2	2	64/64	2	4 DDR2	7
Atom 330	1.60	1	2	32/24	1	4 DDR2	5
Atom D510	1.66	1	2	32/24	1	8 DDR2	1

Table 6.1: Main characteristics of the servers of the local cluster. The total core count is 178 for 40 servers of 10 different SCs.

6.4 Methodology

Server systems: We evaluated Paragon on a 40-machine local cluster (Table 6.1) and a 1000-machine cluster with 14 server types on EC2. We used exclusive (reserved) server instances, i.e., there is no interference from external workloads. We also verified that no external scheduling decisions or actions such as auto-scaling or migration are performed during the course of the experiments.

Schedulers: We compared Paragon with ARQ to four schedulers. First, *Paragon* without admission control, second, a *heterogeneity-oblivious* scheme that only accounts for interference but not heterogeneity. Third, an *interference-oblivious* scheme and finally, a scheduler that is both heterogeneity and interference-agnostic, and assigns applications to *least-loaded* machines.

Workloads: We used 29 single-threaded, 22 multithreaded, 350 multi-programmed and 12 I/O-bound workloads. We use the full SPEC CPU2006 suite and workloads from PARSEC [40], SPLASH-2 [282], BioParallel [150], Minebench [198] and SPECjbb. For multiprogrammed workloads, we use 350 mixes of 4 applications each [228]. The I/O-bound workloads are data mining applications in Hadoop and Matlab. For scenarios with more than 413 applications we replicated these workloads with equal likelihood and randomized their interleaving.

Workload scenarios: For the *small-scale* experiments we examine three workload

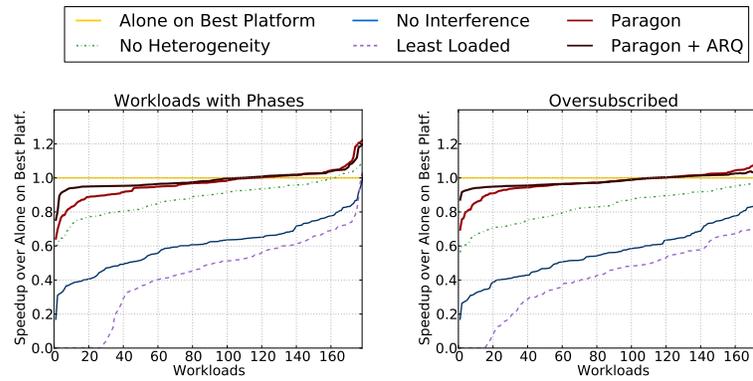


Figure 6.3: Performance comparison of Paragon and ARQ, across two workload scenarios, against Paragon without admission control, a heterogeneity-oblivious, an interference-oblivious and a least-loaded scheduler.

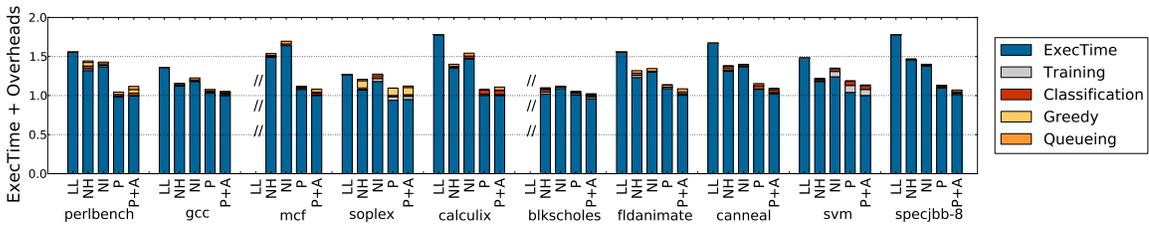


Figure 6.4: Overheads from classification, queuing and scheduling compared to useful execution time. Overall, the overheads in Paragon with ARQ are less than 5% for most applications.

scenarios. First, we examine a *low-load* scenario with 178 applications, selected randomly from the workload pool, and submitted with 10 sec inter-arrival times. Second, a *high-load* scenario where 178 applications arrive following a Gaussian distribution ($\mu=10, \sigma^2=1$) that experience significant *phases* during their execution. Finally, we examine a scenario, where 178 applications arrive with 1 sec intervals. This is an *oversubscribed* scenario, since after a few seconds there are not enough resources to execute all applications concurrently. For the *large-scale* experiments on EC2 we examine an oversubscribed scenario where 7,500 workloads arrive with 1 sec intervals and an additional 1,000 applications arrive in burst after the first 3,750 workloads.

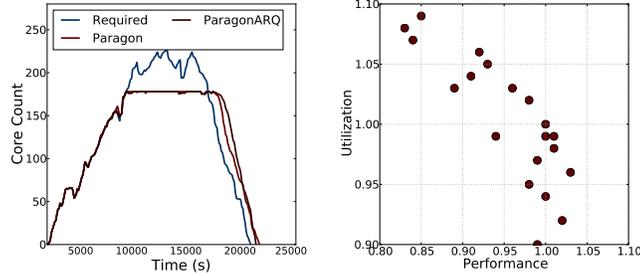


Figure 6.5: Required versus allocated core count for the oversubscribed scenario in the small-scale system and sensitivity of ARQ to the number of queues. Performance and utilization are normalized to the values for 10 queues.

6.5 Evaluation

6.5.1 Small-scale Experiments

Performance: Figure 6.3 shows the performance comparison between the different schedulers for the second and third scenarios in the small-scale cluster. The differences for the low-load scenario where resources are plentiful are small. We focus on the differences between Paragon without and with the use of ARQ. Applications are ordered from worst to best performing. For the scenario with workload phases the applications that preserve their QoS increase from 66% to 91%, and the average performance improves to 99.3%. For the oversubscribed system, while without ARQ only 64% of applications maintain their QoS, with ARQ 88% of workloads preserve their performance requirements. This shows that accounting for resource quality at admission point drains the backlog of queued workloads much faster.

Overheads: ARQ limits waiting time to preserve QoS. Figure 6.4 shows the breakdown of execution time for selected applications in the oversubscribed scenario. Time is divided in useful execution time, overheads from training and classification, overheads from the greedy server selection [76] and overheads from queueing. *mcf* and *blackscholes* do not have a bar for the least-loaded (LL) scheduler because they did not complete successfully due to memory exhaustion in the server. In all cases overheads are very low and execution time for most workloads is very close to one (optimal). The overheads from queueing are less than 5% at all times. The cases where queueing

is high correspond to workloads that had to be diverged to queues of lower resource quality, in which case useful execution time is also suboptimal.

Resource allocation: Figure 6.5a shows the required versus allocated core count for Paragon with and without ARQ for the oversubscribed scenario. Once the system enters the oversubscribed phase ([9000-17000]sec), Paragon without ARQ allocates all available cores and then queues applications, while Paragon with ARQ will only dispatch applications if an appropriate server is freed. This drains the backlog faster since, even though applications are queued for longer, they run in higher quality platforms.

Server utilization: We also measure server utilization before and after the use of ARQ. We focus on the oversubscribed scenario where ARQ has the highest impact. Paragon without ARQ improves utilization by 47% compared to a LL scheduler. Adding ARQ slightly reduces this improvement since applications are queued instead of being dispatched immediately. Despite this, utilization still improves by 45.5%. This means that the performance benefits of ARQ do not incur an efficiency penalty.

Sensitivity to design parameters: Figure 6.5b shows the performance - utilization tradeoff for different numbers of queues. Both metrics are normalized to the values for 10 queues. More queues result in fewer cases of workloads being blocked behind demanding applications, therefore they improve performance, but reduce the number of servers in the corresponding pools, hurting utilization. In contrast, few queues revert to the default scheduler where many applications are scheduled in FIFO order, increasing utilization and hurting performance. 10 queues achieve both high performance and efficiency.

Additional policies: Finally, we evaluate ARQ when computation time and priorities are taken into account in the admission control. Table 6.2 shows the *harmonic mean* (hmean) and *standard deviation* of the difference between the expected and achieved computation time when ARQ implements Shortest Job First (SJF). Workloads are grouped based on their ideal computation time from most short-running to most long-running. SJF prioritizes short over long running applications for all workload classes, with the additional constraint that long applications should still maintain their QoS, therefore cannot be indefinitely bypassed. Results are shown for

Workloads	<i>hmean</i>	<i>standard deviation</i>
Shortest 5%	0.20%	0.15%
Shortest 10%	0.30%	0.08%
Shortest 25%	1.20%	0.30%
Shortest 50%	1.45%	0.34%
Shortest 75%	1.78%	0.26%
Shortest 90%	2.31%	0.55%
Shortest 95%	2.32%	0.57%
All	2.28%	0.53%

Table 6.2: Deviation between expected and achieved *computation time* for workloads in the oversubscribed scenario when ARQ implements SJF. Applications are ranked by increasing expected computation time.

Workloads	<i>hmean</i>	<i>standard deviation</i>
High-priority (20%)	0.80%	0.06%
Low-priority (80%)	2.74%	0.32%

Table 6.3: Deviation between expected and achieved *completion time* for workloads in the oversubscribed scenario when ARQ implements priorities. Applications are grouped in high priority and low priority ones.

the oversubscribed scenario. As shown in the table, short running jobs experience minimal performance degradation, while the long running applications have higher degradations, but still within their QoS requirements (5% execution time increase).

Additionally, we evaluate ARQ in the presence of workload priorities. We increase the priorities of 20% of workloads in the oversubscribed scenario and compare the expected and achieved completion time for them (see Table 6.3). As seen in the table, the high-priority workloads complete within 2% of their ideal completion time. Low-priority applications also complete within their QoS constraints, in most cases, but experience higher performance degradations than high priority workloads.

Large-scale experiments: Figure 6.6 compares the performance of the different schedulers for the large-scale scenario. While Paragon without ARQ only preserves QoS for 61% of workloads, introducing admission control increases that fraction to 83%. Additionally, it bounds degradation to less than 10% for 99% of workloads. This shows that the protocol scales well with the number of servers and applications,

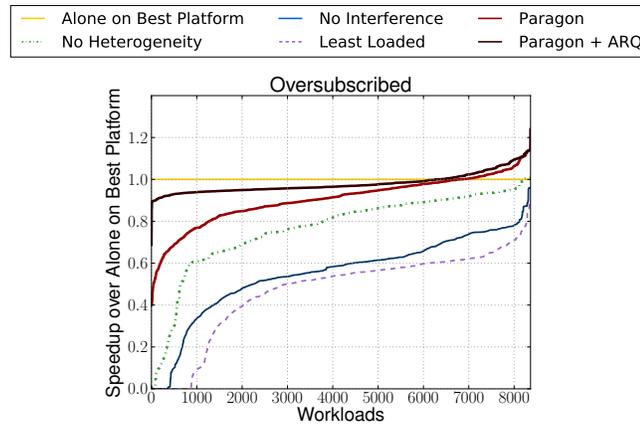


Figure 6.6: Performance for the different schedulers in the oversubscribed scenario on 1,000 EC2 machines.

while maintaining overheads similar to the ones for the small-scale experiments.

6.6 Related Work

We discuss work related to ARQ in terms of admission control in computer systems and analysis of multi-class queueing networks.

Admission control systems: A lot of work has highlighted the importance of admission control in computer systems, including datacenters (DCs). Cherkasova et al. [59, 60] propose a predictive and a session-based admission control scheme respectively for overloaded web servers. The schemes monitor the utilization and QoS achieved at runtime and preemptively adjust the admission policy to more or less aggressive, such that QoS is preserved. In the same spirit, Bartolini et al. [33] propose a self-configurable overload control policy that adjusts the rate of admitted sessions to preserve SLAs and improve utilization. Liu et al. [178] propose an adaptive scheme based on queueing theory to control the performance of multi-tier web applications. Carlstrom et al. [47] also design a session-based admission control protocol for web servers that leverages generalized processor sharing (GPS) [211] to maximize a reward function that corresponds to the rate of completed jobs. Similarly, Salehi et al. [227] propose a preemption-aware admission control system for virtualized systems, where

the system services both internal and external requests, with the internal requests having preemptive priority over external requests. The scheme maximizes the rate of admitted requests, subject to preserving per-application SLAs. Cheng et al. [58] also divide the application space to high and low-priority workloads and partition the server's capacity to service workloads with different priorities. The authors propose a threshold-based admission control algorithm where thresholds depend on the application's priority, and rewards are higher for critical versus non-critical applications. Finally, Guitart et al. [128] consider the problem of admission control in the context of a secure web application and propose an adaptive overload control strategy based on SSL connection differentiation.

Techniques such as predictive admission control [59, 178], protection against DoS attacks, or schemes that additionally account for application security at admission control [128], are orthogonal to the design of ARQ, and can be incorporated in the scheme if the corresponding functionality is desired.

Multi-class queueing networks: Multiclass queueing networks have applications in a wide spectrum of systems ranging from banks, to product lines and network systems. Miller [192] analyzes a multi-class queueing network that optimizes the rewards obtained by accepting or rejecting customers in a system with multiple customer classes. Bertsimas et al. [39] study the distribution of steady-state queue lengths for a multi-class markovian queueing network and propose a methodology based on Lyapunov functions for the performance analysis of MCs with infinite states, including multi-class queueing networks. Kulkarni et al. [164] examine an admission control protocol for multi-class traffic with service priorities in high-speed networks. They assign different size buffers to each class and derive policies to guarantee per-class QoS. Stolyar [251] discusses the stability of multi-class queueing networks, whose stochastic process is a continuous time MC. He shows that the sequence of underlying stochastic processes converges to a fluid process with sample paths defined as fixed points of a special operator and defines the conditions under which the network is stable. In the same context, Chen [56] studies the fluid approximation and stability of a multi-class queueing network.

Gurvich [133] provides an overview of the design and control of multi-class queueing networks (M/M/N queues with multiple types of customers and many servers). He analyzes the V-Model of skills-based routing, and examines how different customer classes are scheduled to servers and how many servers are required to minimize staffing and waiting costs. Sethuraman et al. [235] propose that globally optimal scheduling for a multi-class system with parallel queues reduces to finding the optimal routing matrix under the assumption that the optimal sequencing strategy for each server is a simple static priority policy. Atar et al. [22] also consider asymptotic optimality in a multi-class queueing system with many exponential servers, under the presence of heavy traffic.

In the context of computer systems, Gemikonakli et al. [111] model the performance of a virtualized server using a multi-class M/M/1 queueing model, where applications of different rates arrive in each queue. They analyze the stability, backlog and throughput of the system using an MC model. In a system that resembles a multi-class queue, Yolken et al. [286] propose a game-based capacity allocation system, where each client receives service rate proportional to the bid on resources he submitted to the system operator. Each client has a flow of jobs and although applications are serviced in a FCFS manner, service rates vary across jobs.

6.7 Conclusions

We have presented ARQ, a QoS-aware admission control protocol for heterogeneous datacenters. ARQ divides applications to classes based on their resource quality requirements and queues them separately in a multi-class network. ARQ is derived from validated queueing models, and it improves system throughput by reducing application waiting time, and diverging workloads to different queues when necessary. In an oversubscribed scenario with 8,500 applications on 1,000 servers, 99% of workloads experience less than 10% degradation compared to 79% of workloads without ARQ.

Chapter 7

Tarcil: Reconciling Scheduling Speed and Quality in Large Shared Clusters

7.1 Introduction

In the previous chapters we have presented practical systems that accurately determine the resource requirements of new, previously-unknown cloud applications. Once these requirements are identified, the cluster scheduler must decide where in a cluster to place a new workload. The large size of these clusters (up to tens of thousands of servers) and the high arrival rate of jobs (up to millions of tasks per second) make cluster scheduling quite challenging. The scheduler must determine which specific hardware resources, e.g., servers and cores, should be used by each job. Ideally, scheduling decisions lead to three desirable properties. First, each workload receives resources that enable it to achieve *predictably high performance*. Second, jobs are packed tightly on available servers, achieving *high cluster utilization*. Third, decisions introduce *minimal scheduling overheads*, allowing the scheduler to handle large clusters and high job arrival rates.

Recent research on cluster scheduling can be examined along two dimensions; *scheduling concurrency (throughput)* and *scheduling speed (latency)*.

With respect to scheduling concurrency, there are two groups of work. In the first scheduling is serialized, with a centralized scheduler making all decisions [149, 79]. In the second group, decisions are parallelized through two-level, distributed or shared-state designs. Two-level schedulers, such as Mesos and YARN, use a centralized coordinator to divide resources between frameworks like Hadoop and MPI [139, 267]. Each framework uses its own scheduler to assign resources to incoming tasks. Since neither the coordinator nor the framework schedulers have a complete view of the cluster state and all task characteristics, scheduling is suboptimal [232]. Shared-state schedulers like Omega [232] allow multiple schedulers to concurrently access the whole cluster state using atomic transactions. Finally, Sparrow uses multiple concurrent, stateless schedulers to sample and allocate resources [209].

With respect to the speed at which scheduling decisions happen, there are again two groups of work. The first group examines most of (or all) the cluster state to determine the most suitable resources for incoming tasks, in a way that addresses the performance impact of *hardware heterogeneity* and *interference in shared resources* [76, 120, 186, 200, 285, 296]. For instance, Quasar [79] uses classification to determine the resource preferences of incoming jobs. Then, it uses a greedy scheduler to search the cluster state for resources that meet the application's demands on servers with minimal contention. Similarly, Quincy [149] formulates scheduling as a cost optimization problem that accounts for preferences with respect to locality, fairness and starvation-freedom. Such schedulers make high quality decisions that lead to high application performance and high cluster utilization. Unfortunately, they need to greedily inspect the cluster state on every scheduling event. Their decision overhead can be prohibitively high for large clusters, and in particular for the very short jobs of real-time analytics (100ms - 10s) [209, 288]. Using multiple greedy schedulers improves scheduling throughput but not latency, and terminating the greedy search early typically lowers the decision quality, especially at high cluster loads.

The second group improves the speed of each scheduling decision by only examining a small number of machines. Sparrow reduces scheduling latency through resource sampling [209]. The scheduler examines the state of two randomly-selected servers for each required core and selects the one that becomes available first. While Sparrow

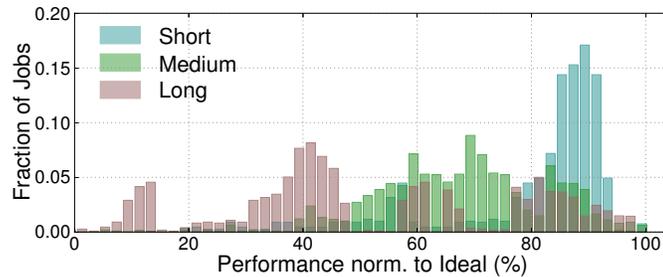


Figure 1a: Sampling-based scheduling.

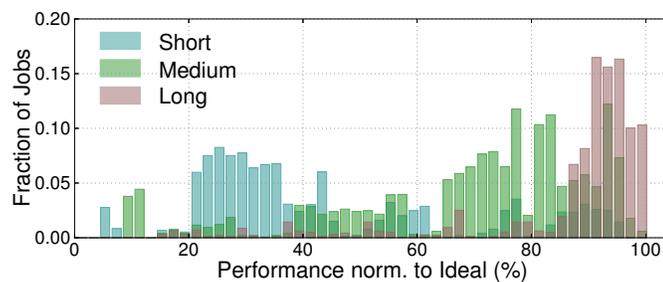


Figure 1b: Centralized scheduling.

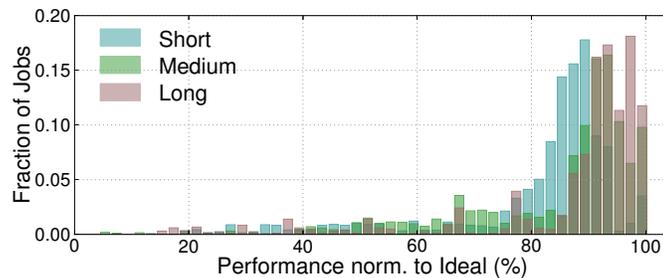


Figure 1c: Tarcil.

Figure 7.1: Distribution of job performance on a 200-server cluster with concurrent, sampling-based [209] and centralized greedy [76] schedulers and Tarcil for three scenarios: 1) short, homogeneous Spark [288] tasks (100ms average duration), 2) Spark tasks of medium duration (1s–10s), and 3) long Hadoop analytics tasks (10s–10min). The ideal performance (100%) assumes no scheduling overheads and no performance degradation due to interference. The cluster utilization is 80%.

improves scheduling speed, its decisions can be poor because it ignores the heterogeneity and interference preferences of jobs. Typically concurrent schedulers follow sampling schemes, while centralized systems are paired with sophisticated scheduling algorithms.

Figure 7.1 illustrates the tradeoff between scheduling speed and quality. Figure 7.1a shows the probability distribution function (PDF) of application performance for three scenarios of variable job duration using Sparrow [209] on a 200-server EC2 cluster. For very short jobs (100ms), fast scheduling allows most workloads to achieve 80% to 95% of the ideal performance on this cluster. In contrast, jobs with medium (1s–10s) or long duration (10s–1min) suffer significant degradation and achieve 50% to 30% of ideal performance. As duration increases, jobs become more heterogeneous in their resource requirements (e.g., preference for high-end cores), and interference between jobs sharing a server matters. In contrast, the scheduling decision speed is not as critical.

Figure 7.1b shows the PDF of job performance using the Quasar scheduler that accounts for heterogeneity and interference [79]. The centralized scheduler leads to near-optimal performance for long jobs. In contrast, medium and short jobs are penalized by the latency of scheduling decisions, which can exceed the execution time of the shortest jobs. Even if we use multiple schedulers to increase the scheduling throughput [232], the per-job overhead remains prohibitively high.

We propose *Tarcil*, a scheduler that achieves the best of both worlds: *high quality and high speed* decisions, making it appropriate for large, highly-loaded clusters that host both short and long jobs. Similar to Quasar [76, 79], Tarcil starts with rich information on the resource preferences and interference sensitivity of incoming jobs. Similar to Sparrow [209], it uses sampling to avoid examining the whole cluster state on every decision. However, there are two key differences in Tarcil’s architecture. First, Tarcil uses sampling not merely to find available resources but to identify resources that best match a job’s resource preferences. The sampling scheme is derived using analytical methods that provide statistical guarantees on the quality of scheduling decisions. Tarcil additionally adjusts the sample size dynamically based on the quality of available resources. Second, Tarcil uses admission control to avoid

scheduling a job that is unlikely to find appropriate resources. To handle the tradeoff between long queueing delays and suboptimal allocations, Tarcil uses a small amount of coarse-grain information on the quality of available resources.

We use two clusters of 100 and 400 servers on Amazon EC2 to show that Tarcil leads to low scheduling overheads and predictably high performance for a wide range of workload scenarios. For a heavily-loaded, heterogeneous cluster running short Spark jobs, Tarcil improves average performance by 41% over Sparrow [209], with some jobs running 2-3x faster. For a cluster running a wide range of applications from short Spark tasks to long Hadoop jobs and low-latency services, Tarcil achieves near-optimal performance for 92% of jobs, in contrast with only 22% of jobs with a distributed, sampling-based scheduler and 48% with a centralized greedy scheduler [79]. Finally, Figure 7.1c, shows that Tarcil enables close to ideal performance for the vast majority of jobs of the three scenarios.

7.2 Background

Our work draws from related efforts to improve scheduling speed and quality in large, shared datacenters:

Concurrent scheduling:

Scheduling becomes a bottleneck for clusters with thousands of servers and high workload churn. An obvious solution is to schedule multiple jobs in parallel [139, 232]. We assume a structure similar to Google’s Omega [232], where multiple scheduling agents can access the whole cluster state. As long as these agents rarely attempt to assign work to the same servers (infrequent conflicts), they proceed concurrently without additional delays. Section 7.5 discusses conflict resolution.

Sampling-based scheduling:

Based on results from randomized load balancing [193, 212], we can design sampling-based cluster schedulers [52, 91, 209]. Sampling the state of just a few servers reduces

the latency of scheduling decisions and the probability of conflicts between concurrent scheduling agents, and is likely to find available resources in lightly- or medium-loaded clusters. The recently-proposed Sparrow scheduler uses *batch sampling* and *late binding* [209]. Batch sampling examines the state of two servers for each of m required cores by an incoming job and selects the m best cores. If the selected cores are busy, tasks are queued locally in the sampled servers and assigned to the machine where resources become available first. In our evaluation we compare Tarcil with Sparrow.

Heterogeneity & interference-aware scheduling:

Hardware heterogeneity occurs in large clusters because servers are populated and replaced over time [76, 285]. Moreover, the performance of tasks sharing a server may degrade significantly due to interference on shared resources such as caches, memory and I/O channels [76, 120, 186, 204]. A scheduler can improve task performance significantly by taking into consideration its resource preferences. For instance, a particular task may perform much better on 2.3GHz Ivy-Bridge cores compared to 2.6GHz Nehalem cores, while another task may be particularly sensitive to interference from cache-intensive workloads executing on the same server.

The key challenge in heterogeneity and interference-aware scheduling is knowing the preferences of incoming jobs. We start with a system like Quasar that automatically estimates resource preferences and interference sensitivity [76, 79]. Quasar profiles each incoming job for a few seconds on two server types, while two microbenchmarks place pressure on two shared resources. The sparse profiling signal on resource preferences is transformed into a dense signal using collaborative filtering [43, 161, 219, 281]. Collaborative filtering projects the signal against all information available from previously-run jobs, identifying similarities in resource and interference preferences. These include examples such as the preferred core frequency and cache size for a job or the memory and network contention it generates. Quasar performs profiling and collaborative filtering online. We perform this analysis offline, given that workloads like real-time analytics are repeated multiple times, potentially over different data (e.g., daily or hourly).

7.3 The Tarcil Scheduler

7.3.1 Overview

Tarcil is a shared-state scheduler that allows multiple, concurrent agents to operate on the cluster state [232]. In this section, we describe the operation of a single agent.

The scheduler processes incoming workloads as follows. Upon submission, Tarcil first *looks up the job's resource and interference sensitivity preferences* [76, 79]. This information provides estimates of the relative performance on the different server platforms, as well as estimates of the interference the workload can tolerate and generate in shared resources (caches, memory, I/O channels). Next, Tarcil performs *admission control*. Given statistics on the cluster state, it determines whether the scheduler is likely to quickly find resources of satisfactory quality for a job, or whether it should queue it for a while. Admission control is useful when the cluster is highly loaded. A queued application waits until it has a high probability of finding appropriate resources or until a queueing-time threshold is reached. Tarcil maintains coarse-grained statistics on available resources for admission control decisions. These statistics are updated as jobs begin and end execution.

For admitted jobs, Tarcil performs sampling-based scheduling with the sample size adjusted to satisfy statistical guarantees on the quality of allocated resources. The scheduler also uses batch sampling if a job requests multiple cores. Tarcil examines the quality of sampled resources to select those best matching the job's preferences. It additionally monitors the performance of running jobs. If a job runs significantly below its expected performance, the scheduler adjusts the scheduling decisions. This is useful for long-running workloads; for short jobs, the initial scheduling decision determines performance with little space for adjustments.

7.3.2 Analytical Framework

We use the following framework to design and analyze sampling-based scheduling in Tarcil.

Resource unit (RU): Tarcil manages resources at RU granularity using Linux

containers [66]. Each RU consists of one core and an equally partitioned fraction of the server’s memory and storage capacity and the provisioned network bandwidth. For example, a server with 16 cores, 64GB DRAM, 480GB of Flash and a 10GE NIC has 16 RUs, each with 1 core, 4 GB DRAM, 30GB of Flash and 625ME of network bandwidth. Because all our experiments are on public cloud providers where the network topology is unknown, in our evaluation we do not partition network bandwidth.

RU quality: The utility an application can extract from an RU depends on the hardware type (e.g., 2GHz vs 3GHz core) and the interference on shared resources from other jobs on the same server. Classification [76, 79] obtains the interference preferences of an incoming job using a small set of microbenchmarks to inject pressure of increasing intensity (from 0 to 99%) on one of ten shared resources of interest [74]. Interference preferences capture, first, the amount of pressure t_i a job can tolerate in each shared resource i ($i \in [1, N]$), and second, the amount of pressure c_i it itself will generate in the same resource. High values of t_i or c_i imply that a job will tolerate or cause a lot of interference on resource i . t_i and c_i take values in $[0, 99]$. In most cases, jobs that cause a lot of interference in a resource are also sensitive to interference on the same resource. Hence, to simplify the rest of the analysis we assume that $t_i = 99 - c_i$ and express resource quality as a function of caused interference in an RU.

Let W be an incoming job and V_W the vector of interference it will cause in the N shared resources, $V_W = [c_1, c_2, \dots, c_N]$. To capture the fact that different jobs are sensitive to interference on different resources [186], we reorder the elements of V_W by decreasing value of c_i and get $V'_W = [c_j, c_k, \dots, c_n]$, with $c_j > c_k > \dots > c_n$. Finally, we obtain a single value for the resource requirements of W using an order-preserving encoding scheme that transforms V'_W to a concatenation of its elements:

$$V_{W_{enc}} = c_j \cdot 10^{(2 \cdot (N-1))} + c_k \cdot 10^{(2 \cdot (N-2))} + \dots + c_n \quad (7.1)$$

For example if $V'_W = [84, 31]$ then $V_{W_{enc}} = 8431$. The expression above is provably the most dense encoding that preserves the full entropy of the values of vector V'_W and their ordering, for general V'_W . Finally, for simplicity we normalize $V_{W_{enc}}$ in $[0, 1]$

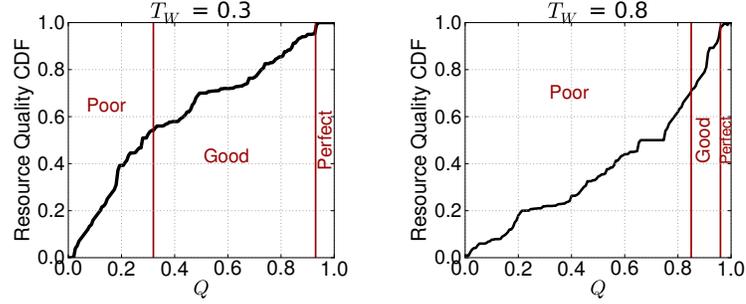


Figure 7.2: Distribution of resource quality Q for two workloads with $T_W = 0.3$ (left) and $T_W = 0.8$ (right).

and derive the target resource quality for job W :

$$T_W = \frac{V_{W_{enc}}}{10^{2N} - 1}, T_W \in [0, 1] \quad (7.2)$$

A high value for the quality target T_W implies that job W is resource-intensive. Its performance will depend a lot on the quality of the scheduling decision.

We now need to find RUs that closely match this target quality. To determine if an available resource unit H is appropriate for job W , we calculate the interference caused on this RU by all other jobs occupying RUs on the same server. Assuming M resource units in the server, the total interference H experiences on resource i is:

$$C_i = \frac{\sum_{m \neq H} C_i}{M - 1} \quad (7.3)$$

Starting with vector $V_H = [C_1, C_2, \dots, C_N]$ for H and using the same reordering and order-preserving encoding as for T_W , we calculate the quality of resource H as:

$$U_H = 1 - \frac{V_{H_{enc}}}{10^{2N} - 1}, U_H \in [0, 1] \quad (7.4)$$

The higher the interference from colocated tasks, the lower U_H will be. Resources with low U_H are more appropriate for jobs that can tolerate a lot of interference and vice versa.

Comparing U_H for an RU against T_W allows us to judge the quality of resource H for incoming job W :

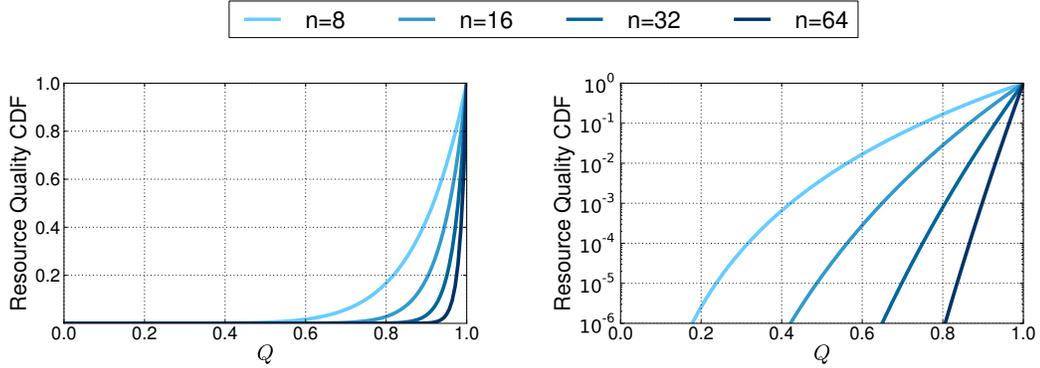


Figure 7.3: Resource quality CDFs under the uniformity assumption in linear and log scale for sample size $R=8, 16, 32$ and 64 .

$$Q = \begin{cases} 1 - (U_H - T_W) & , \text{ if } U_H \geq T_W \\ T_W - U_H & , \text{ if } U_H < T_W \end{cases} \quad (7.5)$$

If Q equals 1, we have an ideal assignment with the server tolerating as much interference as the new job generates. If Q is within $[0, T_W]$, selecting RU H will degrade the job’s performance. If Q is within $[T_W, 1)$, the assignment will preserve the workload’s performance but is suboptimal. It would be better to assign a more demanding job on this resource unit.

Resource quality distribution: Figure 7.2 shows the distribution of Q for a 100-server cluster with ~ 800 RUs (see Section 7.6 for cluster details) and one hundred, 10-min Hadoop jobs as resident load (50% cluster utilization). For a non-demanding new job with $T_W = 0.3$ (left), there are many appropriate RUs at any point in time. In contrast, for a demanding job with $T_W = 0.8$, only a small number of resources will lead to good performance. Obviously, the scheduler must adjust the sample size for incoming jobs based on T_W .

7.3.3 Sampling-based Scheduling with Guarantees

We can now derive the sample size that provides statistical guarantees on the quality of scheduling decisions.

Assumptions and analysis: To make the analysis independent of cluster load, we make Q an absolute ordering of RUs in the cluster. Starting with equation (5),

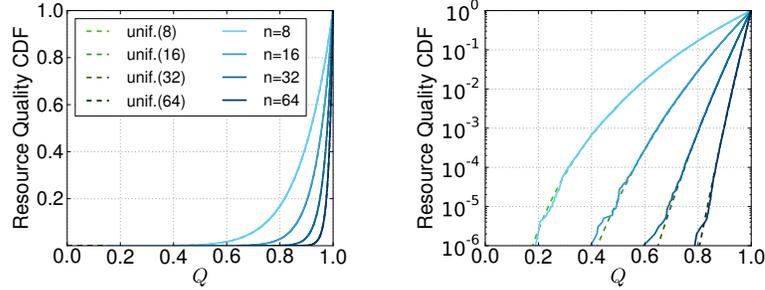


Figure 7.4: Comparison of resource quality CDFs under the uniformity assumption, and as measured in a 100-server cluster.

we sort RUs based on Q for incoming job W , breaking any ties in quality with a fair coin, and distribute them uniformly in $[0, 1]$, i.e., for N_{RU} total RUs, $Q(i) = i/(N_{RU} - 1)$, $i \in [0, N_{RU} - 1]$. Because Q is now a *probability distribution* function of resource quality, we can derive the sample size in the following manner.

Assume that the scheduler samples R RU candidates for each RU needed by an incoming workload. If we treat the qualities of these R candidates as random variables Q_i ($Q_1, Q_2, \dots, Q_R \sim U[0, 1]$) that are *uniformly distributed* by construction and statistically independent from each other (*i.i.d*), we can derive the distribution of quality Q after sampling. The cumulative distribution function (CDF) of the resource quality of each candidate is: $F_{Q_i}(x) = Prob(Q_i \leq x) = x$, $x \in [0, 1]$ ¹. Since the candidate with the highest quality is selected from the sampled set, its resource quality is the random variable $A = \max\{Q_1, Q_2, \dots, Q_R\}$, and its CDF is:

$$\begin{aligned} F_A(x) &= Prob(A \leq x) = Prob(Q_1 \leq x \wedge \dots \wedge Q_R \leq x) \\ &= Prob(Q_i \leq x)^R = x^R, \quad x \in [0, 1] \end{aligned} \quad (6)$$

This implies that the distribution of quality after sampling *only* depends on the sample size R . Figure 7.3 shows CDFs of resource quality distributions under the uniformity assumption, for sample sizes $R = \{8, 16, 32, 64\}$. The higher the value of R , the more skewed to the right the distribution is, hence the probability of finding only candidates of low quality quickly diminishes to 0. For example, for $R = 64$ there

¹This assumes Q_i to be continuous variables, although in practice they are discrete. This makes the analysis independent of the cluster size N_{RU} . The result holds for the discretized version of the equation.

is a 10^{-6} probability that none of the sampled RUs will have resource quality of at least $Q = 80\%$ ($Prob(Q < 0.8 | \forall RU) = 10^{-6}$).

Figure 7.4 *validates the uniformity assumption* on a 100-server EC2 cluster running short Spark tasks (100msec ideal duration) and longer Hadoop jobs (1-10min). The cluster load is 70-75% (see methodology in Section 7.6). In all cases, the deviation between the analytically derived and measured distributions of Q is minimal, which shows that the analysis above holds in practice. In general, the larger the cluster, the more closely the quality distribution approximates uniformity.

Large jobs: For jobs that need multiple RUs, Tarcil uses *batch sampling* [209, 212]. For m requested units, the scheduler samples $R \cdot m$ RUs and selects the m best among them as shown in Figure 7.5a. Some applications experience locality between sub-tasks or benefit from allocation of all resources in a small set of machines (e.g., within a single rack). In such cases, for each sampled RU, Tarcil examines its neighboring resources and makes a decision based on their aggregate quality as shown in Figure 7.5b. Alternatively, if a job prefers distributing its resources across machines the scheduler will allocate RUs in different machines, racks and/or cluster switches, assuming knowledge of the cluster’s topology. Placement preferences for reasons such as security [237] can also be specified in the form of attributes at submission time by the user.

Sampling at high load: Equation (6) estimates the probability of finding near-optimal resources accurately when resources are not scarce. When the cluster operates at high load, we must increase the sample size to guarantee the *same probability* of finding a *candidate of equally high quality*, as when the system is unloaded. Assume a system with $N_{RU} = 100$ RUs. Its discrete CDF is $F_A(x) = P[A \leq x] = x$, $x = 0, 0.01, 0.02, \dots, 1$. For sample size R , this becomes: $F_A(x) = x^R$, and a quality target of $Pr[Q < 0.8] = 10^{-3}$ is achieved with $R = 32$. Now assume that 60% of the RUs are already busy. If, for example, only 8 of the top 20 candidates for this task are available at this point, we need to set R s.t. $Pr[Q < 0.92] = 10^{-3}$, which requires a sample size of $R = 82$. Hence, the sample size for a highly loaded cluster can be quite high, degrading scheduling latency. In the next section, we introduce an admission control scheme that bounds sample size and scheduling latency, while still allocating

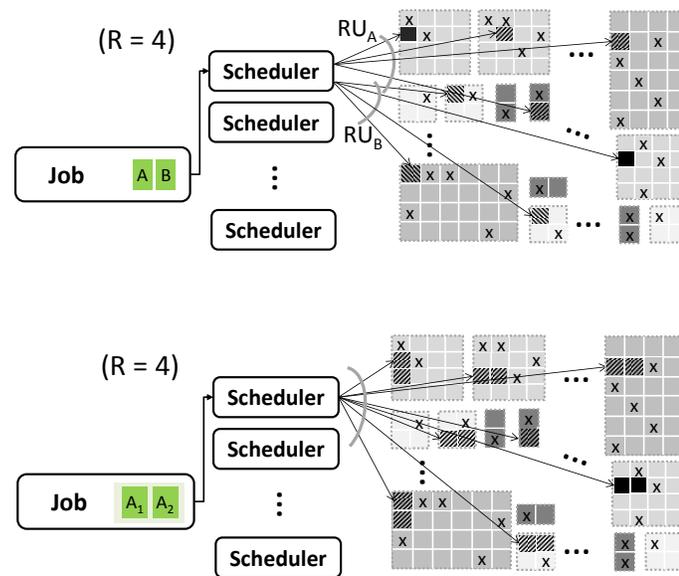


Figure 7.5: Batch sampling in Tarcil with sample size $R = 4$ for (a) a job with two independent tasks A and B , and (b) a job with two subtasks A_1 and A_2 that exhibit locality. x -marked RUs are already allocated, striped RUs are sampled, and solid black RUs are allocated to the incoming job after sampling.

high quality resources.

7.4 Admission Control

7.4.1 Pre-scheduling Queueing

When available resources are plentiful, jobs are immediately scheduled using the sampling scheme described in Section 7.3. However, when load is high, the number of resources of sufficient quality may be very small and the sample size needed to find them can become quite large. Tarcil employs a simple admission control scheme that queues jobs until resources of proper quality become available and estimates how long an application should wait at admission control.

A simple indication to trigger job queueing is the count of available RUs in the cluster. This, however, does not yield sufficient insight into the quality of available RUs. If most RUs have poor quality for an incoming job, it may be better for it to

wait. Unfortunately, a naïve quality check involves accessing the state of the whole cluster, which would introduce prohibitive overheads. Instead, we maintain a small amount of coarse-grain information which allows for a fast check. We leverage the information on contention scores that is already maintained for each RU to construct a contention score vector $[C_1 C_2 \dots C_N]$ from the resource contention C_i it experiences in each of its resources, due to interference from neighboring RUs. We use *locality sensitive hashing* (LSH) based on random selection to hash these vectors into a small set of buckets [19, 53, 219]. LSH computes the cosine distance between vectors and assigns RUs with similar contention scores in the respective resources to the same bucket. We also separate RUs by platform type to account for heterogeneity. We *only* keep a single count of available RUs for each bucket. The hash for an RU (and the counter of the corresponding bucket) needs to be recalculated upon instantiation or completion of a job in an RU. Updating the per-bucket counters is a fast operation, out of the critical path for scheduling. Note that excluding updates in RU status, LSH is only performed once.

Admission control works as follows. We check the bucket(s) that correspond to the resources with quality that matches the incoming job’s preferences. If these buckets have counters close to the number of RUs the job needs, the application is queued. Queued applications wait until the probability that resources are freed increases or until an upper bound for waiting time is reached. To estimate waiting time, Tarcil records the rate at which RUs of each bucket became available in recent history. Specifically, it uses a simple feedback loop to estimate when the probability that an appropriate RU exists approximates 1 for a target bucket. The distribution is updated every time an RU from that bucket is freed. Tarcil also sets an upper bound for waiting time at $\mu + 2 \cdot \sigma$, where μ and σ are the mean and standard deviation of the corresponding “time-until-free” PDF. If the estimated waiting time is less than the upper bound, the job waits for resources to be freed; otherwise it is scheduled to avoid excessive queueing. Although admission control adds some complexity, in practice it only delays workloads at very high cluster utilizations (over 80%-85%).

Validation of waiting time estimation: Figure 7.6 shows the probability that

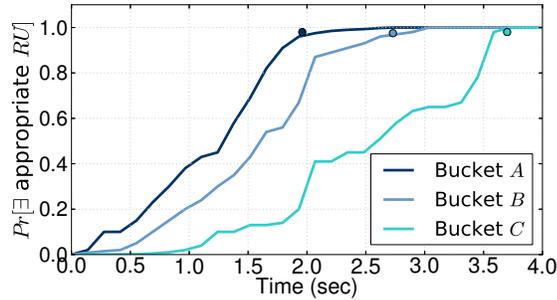


Figure 7.6: Actual and estimated (dot) probability for a target RU to exist as a function of waiting time for three buckets.

a desired RU will become available within time t for different buckets for a heterogeneous 100-server EC2 cluster running short Spark tasks and longer Hadoop jobs. The cluster utilization is approximately 85%. We show the probabilities for r3.2xlarge (8 vCPUs) instances with CPU contention (A), r3.2xlarge instances with network contention (B), and c3.large (2 vCPUs) instances with memory contention (C). The distributions are obtained from recent history and vary across buckets. The dot in each line shows the estimated waiting time by Tarcil, which closely approximates the measured time for an appropriate RU to be freed (less than 8% deviation on average). In all experiments, we use 20 buckets, and history of the past 2 hours, which was sufficient to make accurate estimations of available resources. The number of buckets and/or history length may vary for different systems.

7.4.2 Post-scheduling Queueing

A job that exceeds the upper bound on queueing may still require a high sample size. To avoid excessive scheduling overheads, we cap the sample size at 32 and instead use late binding on the sampled servers until resources become available [209]. If the best two of the 32 sampled RUs are currently busy, the job is locally queued in both until the first RU is freed and is subsequently removed from the queue of the second RU. Note that local queueing is unlikely in practice.

7.5 Tarcil Implementation

7.5.1 Tarcil Components

Figure 7.7 shows the components of the scheduler. Tarcil is a distributed, shared-state scheduler and, unlike Quincy or Mesos, it does not have a central coordinator [139, 149]. Scheduling agents work in parallel, are load-balanced by the cluster front-end, and each agent has a local copy of the shared server state, which contains the list and status of all RUs in the cluster.

Since all schedulers have full access to the cluster state, conflicts are possible. Conflicts between agents are resolved using *lock-free optimistic concurrency* as discussed in [232]. The system maintains one resilient master copy of state. Each scheduling agent has a local copy of this state which is updated frequently. When an agent makes a scheduling decision it attempts to update the master copy of the state using an atomic write operation. While an agent performs this action no other agent can update these resources in the master copy. Once the commit is successful the resources are yielded to the corresponding agent. Any other agent with conflicting decisions needs to resample resources. The local copy of state of each agent is periodically synced (every 5-10sec) with the master. The timing of the updates includes a small random seed such that not all agents update their state at exactly the same time, making the master the bottleneck. When the sample size is small, decisions of scheduling agents rarely overlap and each scheduling action is fast ($\sim 10 - 20\text{ms}$, for a 100-server cluster and $R = 8$, over an order of magnitude faster than centralized approaches). When the number of sampled RUs increases beyond $R = 32$ for very large jobs, conflicts can become more frequent, which we resolve using incremental transactions on the non-conflicting resources [232]. In the event where one scheduling agent crashes, an idle cluster server resumes its role, once it has obtained a copy of the master state.

Each worker server has a *local monitor* module that handles scheduling requests, federates resource usage in the server, and updates the quality of RUs. When a new task is assigned to a server by a scheduling agent, the monitor updates the status of the RU in the master copy and notifies the scheduling agent and admission

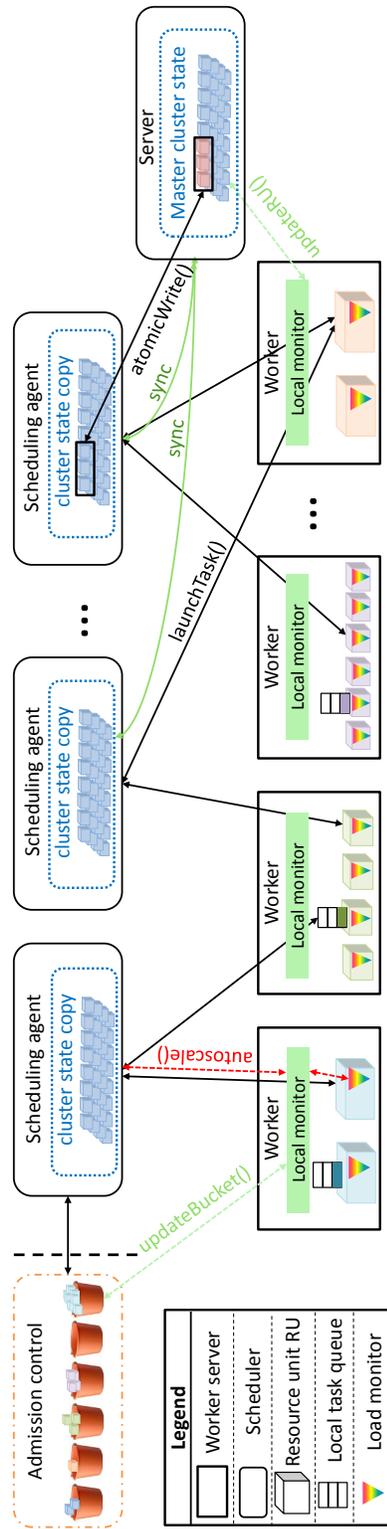


Figure 7.7: The different components of the scheduler and their interactions. Each of the scheduling agents has a local copy of the cluster state, while an additional server has the master copy of the state, for scheduling actions. Each worker server has a local monitor that tracks resource usage, and handles scheduling requests.

control. Finally, a per-RU *load monitor* evaluates performance in real time. When the monitor detects that a job's performance deviates from its expected target, it notifies the proper agent for a possible allocation adjustment. The load monitor also notifies agents of CPU or memory saturation, which triggers resource autoscaling (see Section 7.5.2).

We currently use Linux containers to partition servers into RUs [?]. Containers enable CPU, memory and I/O isolation. Each container is configured to a single core and a fair share of the memory and storage subsystem, and the network bandwidth. Containers can be merged to accommodate multicore workloads, using *cgroups*. Virtual machines (VMs) can also be used to enable workload migration [204, 269, 270, 283], but would incur higher overheads.

Figure 7.8 traces a scheduling event. Once a job is submitted, admission control evaluates whether it should be queued or not. Once the assigned scheduling agent sets the sample size according to the job's constraints, it samples the shared cluster state for the required number of RUs. Sampling happens locally in each agent. The agent computes the resource quality of sampled resources and selects the ones that should be allocated to the job. The actual selection takes into account the resource quality and platform preferences, as well as any locality preferences of a task. The agent then attempts to update the master copy of the state. Upon a successful commit the agent notifies the local monitor of the selected server(s) over RPC and launches the task in the target RU(s). The local monitor notifies admission control, and the master copy to update their state. Once the task completes, the local monitor issues RPCs that update the master state and notify the agent and admission control; the scheduling agent then informs the cluster front-end.

7.5.2 Adjusting Allocations

For short-running tasks, the quality of the initial assignment is particularly important. For long-running tasks, we must also consider the different phases the program can go through [163]. Similarly, we must consider cases where Tarcil makes a suboptimal allocation due to inaccurate classification, deviations from fully random selection in

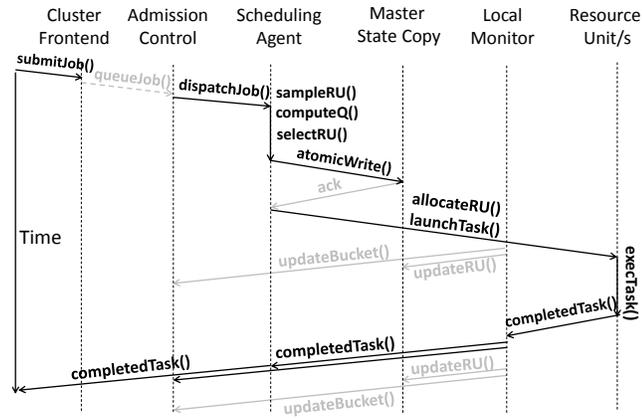


Figure 7.8: Trace of a scheduling event in Tarcil.

the sampling process, or a compromise in resource quality at admission control. Tarcil uses the per-server *load monitor*, i.e., a local daemon running in each RU, to measure the performance of active workloads in real time. This can correspond to instructions per second (IPS), packets per second or a high-level application metric, depending on the application type. Tarcil compares this metric to any performance targets the job provides or are available from previous runs of the same application. If there the job is not satisfying its QoS constraints, the scheduler takes action. Since we are using containers, the primary action we take is to avoid scheduling other jobs on the same server. For scale-out workloads, the system also employs a simple *autoscale* service which allocates more RUs (locally or not) to improve the job’s performance.

7.5.3 Fairness

Users can submit jobs with priorities. Jobs with higher priority will bypass others at admission control and preempt lower-priority jobs during resource selection. Tarcil also allows the user to select between incremental scheduling, where tasks from a job get progressively scheduled as resources become available and all-or-nothing gang scheduling, where either all or no task from a job is scheduled. We leave the experimental evaluation of priorities and other policies to future work.

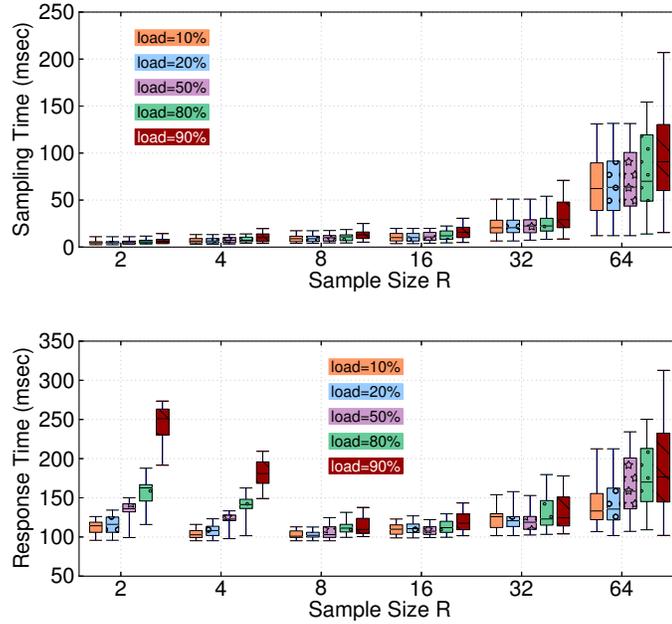


Figure 7.9: Sensitivity of sampling overheads and response times to sample size.

7.6 Evaluation

7.6.1 Tarcil Analysis

We first evaluate Tarcil’s scalability and its sensitivity to parameters such as the sample size and task duration.

Sample size: Figure 7.9 shows the sensitivity of sampling overheads and response times to the sample size for homogeneous Spark jobs with 100msec duration and cluster loads varying from 10% to 90% on the 110-server EC2 cluster. All machines are r3.2xlarge memory-optimized instances (61GB of RAM). 10 servers are used by the scheduling agents, and the remaining 100 machines serve incoming load. The boundaries of the boxplots depict the 25th and 75th percentiles, the whiskers the 5th and 95th percentiles and the horizontal line in each boxplot shows the mean. As sample size increases, the overheads increase. Until $R = 32$ overheads are marginal even at high loads, but they increase substantially for $R \geq 64$, primarily due to the overhead of resolving conflicts between the 10 scheduling agents used. Hence, we cap sample size to $R = 32$ even under high load. Response times are more sensitive to

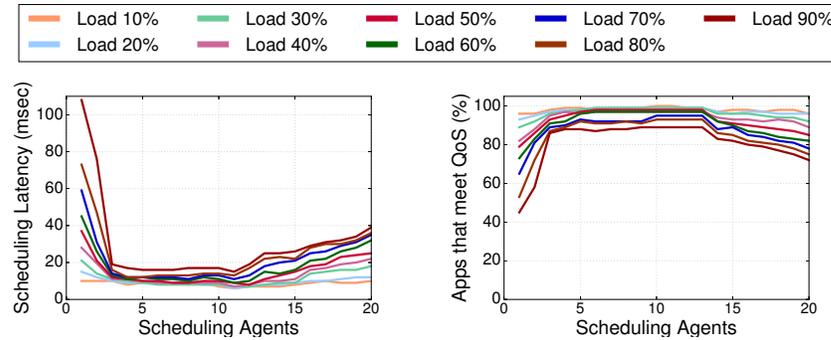


Figure 7.10: Sensitivity to the number of concurrent scheduling agents. Figure 7.10a shows the scheduling latency, and Figure 7.10b the fraction of jobs that meet QoS.

sample size. At low load, high quality resources are plentiful and increasing R makes little difference to performance. As load increases, sampling with $R = 2$ or $R = 4$ is unlikely to find good resources. Sample size of $R = 8$ is optimal for both low and high cluster loads, in this scenario.

Number of scheduling agents: We now examine how the number of agents that perform concurrent scheduling actions affects the quality and latency of scheduling. Figure 7.10a shows how scheduling latency changes as we increase the number of scheduling agents. The cluster load varies again from 10% to 90%, and the load is the same homogeneous Spark tasks with 100msec optimal duration, as before. We set the sample size to $R = 8$, which was the optimal, based on the previous experiment. When the number of schedulers is very small (below 3), latency suffers at high loads due to limited scheduling parallelism. As the number of agents increases latency drops, until 12 agents. Beyond that point, latency slowly increases due to increasing conflicts among agents. For larger cluster sizes, the same number of agents would not induce as many conflicts. Figure 7.10b shows how the fraction of tasks that meet QoS changes as the number of scheduling agents increases. As previously seen, if the number of agents is very small, many jobs experience increased response times. As more agents are added, the vast majority of jobs meet their QoS until high cluster loads. When cluster load exceeds 80%, QoS violations are caused primarily due to queuing at admission control, instead of limited scheduling concurrency. In general, 3 scheduling agents are sufficient to get the minimum scheduling latency; in following

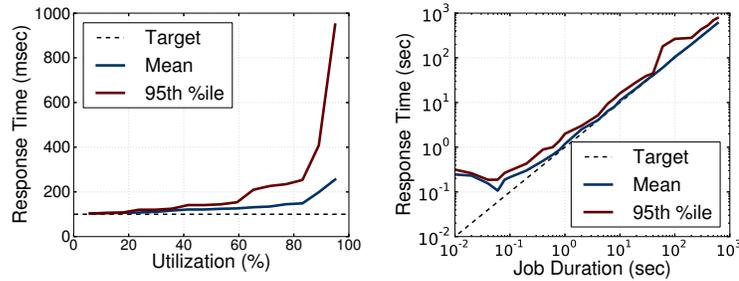


Figure 7.11: Response times when (a) increasing cluster load, and (b) when decreasing task duration with constant load.

comparisons with Sparrow we use 10 agents to ensure a fair comparison, since Sparrow uses a 10:1 worker to agent ratio.

Cluster load: Figure A.3a shows the average and 95th percentile response times when we scale the cluster load in the 110-server EC2 cluster. The incoming jobs are homogeneous Spark tasks with 100msec target duration. We increase the task arrival rate to increase cluster load. The target performance of 100msec includes no scheduling overheads or degradation due to suboptimal scheduling. The reported response times include the task execution time and all overheads. The mean of response times with Tarcil remains almost constant until loads over 85%. At very high loads, admission control and the large sample size increase the scheduling overheads, affecting performance. The 95th percentile is more volatile at high loads, but only exceeds 250msec at cluster loads of 80% or higher. Tasks with very high response times are typically those delayed by admission control until the wait-time threshold is reached. Sampling itself adds marginal overheads until 90% load. At very high loads scheduling overheads are dominated by queueing time and increased sample sizes.

Task duration: Figure A.3b shows the average and 95th percentile response times as a function of task duration, which ranges from 10msec to 600sec. The cluster load is 80% in all cases. For long tasks the mean and 95th percentile closely approximate the target performance. When task duration is below or close to 100msec, the scheduling overhead dominates. Despite this, the mean and 95th percentile remain very close, which shows that performance unpredictability is limited. For long jobs, configuring and allocating large amounts of resources dominates the scheduling overheads, while

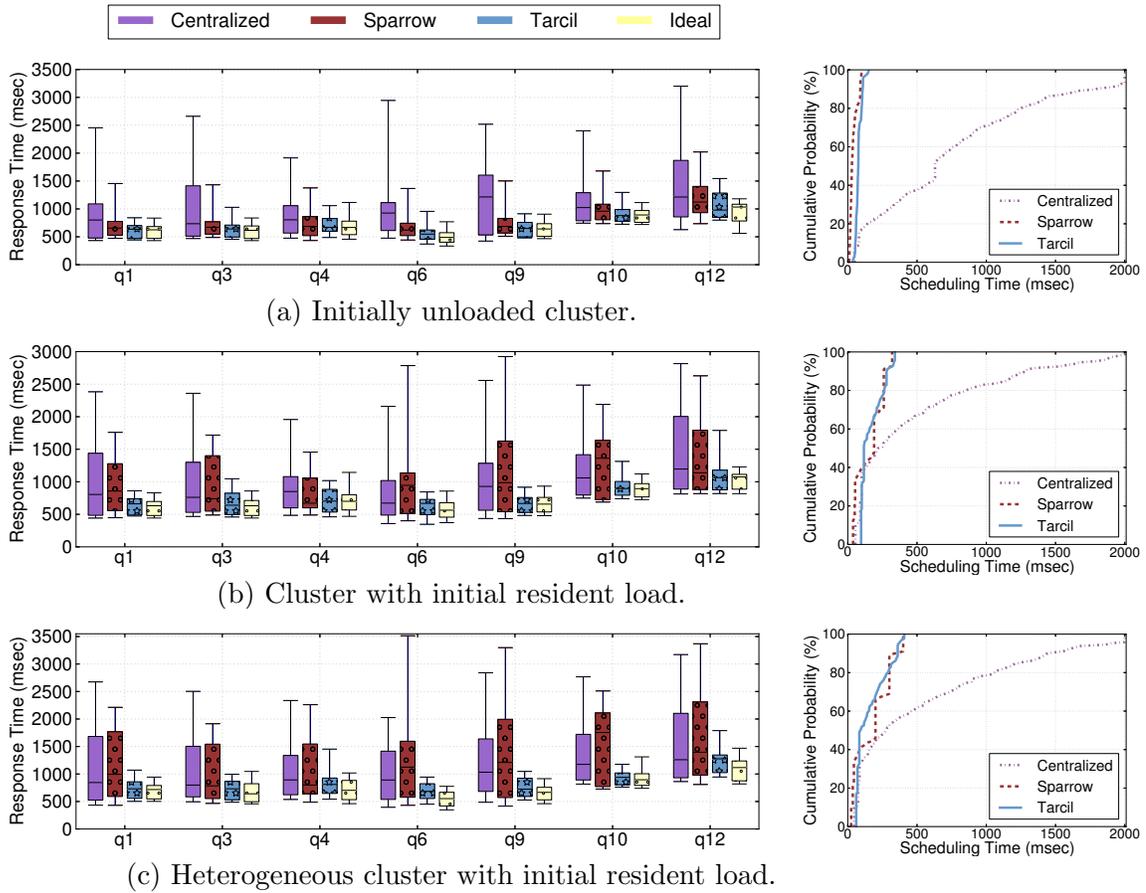


Figure 7.12: Response times for different TPC-H query types (left) and CDFs of scheduling overheads (right).

for large numbers of short tasks, queueing delay dominates.

7.6.2 Comparison with Other Schedulers

Methodology: We compare Tarcil to Sparrow [209] and Quasar [79]. Sparrow uses multiple scheduling agents and sampling ratio of $R = 2$ servers for every core required, as recommended in [209]. Quasar has a centralized greedy scheduler that searches the cluster state with a scheduling timeout of 2 seconds. Sparrow does not take into account heterogeneity or interference preferences for incoming jobs, while Tarcil and Quasar do. We evaluate these schedulers on the same 110-server EC2 cluster with

r3.2xlarge memory-optimized instances (61GB of RAM). 10 servers are dedicated to the scheduling agents for Tarcil and Sparrow and a single server for Quasar. While we could replicate Quasar’s scheduler for fault tolerance, it would not help with the latency of each scheduling decision. Additionally, Quasar schedules applications at job, not task, granularity (when applicable), which reduces its scheduling load. Unless otherwise specified, Tarcil uses sample sizes of $R = 8$ during low load.

TPC-H workload

We compare the three schedulers on the TPC-H decision support benchmark. TPC-H is a standard proxy for ad-hoc, low-latency queries that comprise a large fraction of load in shared clusters. We use a similar setup as the one used to evaluate Sparrow [209]. TPC-H queries are compiled into Spark tasks using Shark [94], a distributed SQL data analytics platform. The Spark plugin for Tarcil is 380 lines of code in Scala. Each task triggers a scheduling request for the distributed schedulers (Tarcil and Sparrow), while Quasar schedules jointly all tasks from the same computation stage. We constrain tasks in the first stage of each query to the machines holding their input data (3-way replication). All other tasks are unconstrained. We run each experiment for 30 minutes, with multiple users submitting randomly-ordered TPC-H queries to the cluster. The results discard the initial 10 minutes (warm-up) and capture a total of 40k TPC-H queries and approximately 134k jobs. Utilization at steady state is 75-82%.

Unloaded cluster: We first examine the case where TPC-H is the only workload present in the cluster. Figure 7.12a shows the response times for seven representative query types [293]. Response times include all scheduling overheads from sampling or the greedy selection, and queueing. Boundaries show 25th and 75th percentiles and whiskers the 5th and 95th percentiles. The ideal scheduler corresponds to a system that identifies the resources of optimal quality (including heterogeneity and interference preferences) with zero delay. Figure 7.12a shows that the centralized scheduler experiences the highest variability in performance. Although some queries complete

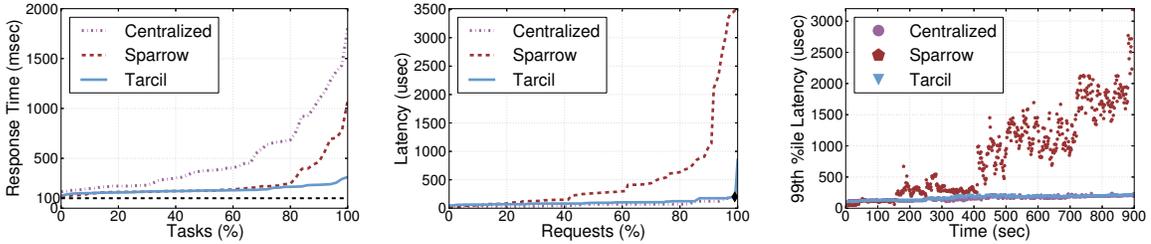


Figure 7.13: Performance of scheduled Spark tasks and resident memcached load (aggregate and over time).

very fast because they receive high quality resources, most experience high scheduling delays. To verify this, we also show the scheduling time CDF on the right of Figure 7.12a. While Tarcil and Sparrow have tight bounds on scheduling overheads, the centralized scheduler adds up to 2 seconds of delay (timeout threshold). Comparing the query performance using Sparrow and Tarcil, we see that the difference is small, 8% on average. Tarcil approximates the ideal scheduler more closely, as it accounts for each task’s resource preferences. Additionally, Tarcil constrains performance unpredictability. The 95th percentile is reduced by 80%-2.4x compared to Sparrow.

Cluster with resident load: The difference in scheduling quality becomes more clear when we introduce cross-application interference. Figure 7.12b shows a setup where 40% of the cluster is busy servicing background applications, including other Spark jobs for machine learning processing, long Hadoop workloads, and latency-critical services like memcached. These jobs are *not* being scheduled by the examined schedulers. While the centralized scheduler still adds considerable overhead to each job (Figure 7.12b, right), its performance is now comparable to Sparrow. Since Sparrow does not account for sensitivity to interference, the response time of queries that experience resource contention is high. Apart from average response time, the 95th percentile also increases significantly (poor predictability). In contrast, Tarcil accounts for resource preferences and only places tasks on machines with acceptable interference levels. It maintains an average performance only 6% higher compared to the unloaded cluster across query types. More importantly, it preserves the low performance jitter by bounding the 95th percentile of response times.

Heterogeneous cluster with resident load: Next, in addition to interference, we also introduce hardware heterogeneity. The cluster size remains constant but 75% of the worker machines are replaced with less or more powerful servers, ranging from general purpose medium and large instances to quadruple compute- and memory-optimized instances. Figure 7.12c shows the new performance for the TPC-H queries. As expected, response times increase, since some of the high-end machines are replaced by less powerful servers. More importantly, performance unpredictability increases when the resource preferences of incoming jobs are not accounted for. In some cases ($q9$, $q10$), the centralized scheduler now outperforms Sparrow despite its much higher scheduling overheads. Tarcil preserves response times close to those in the unloaded cluster and very close to those achieved with the ideal scheduler.

Impact on Resident Memcached Load

Finally, we examine the impact of scheduling decisions on resident cluster load. In the same heterogeneous cluster (110 nodes on EC2, 100 workers and 10 schedulers), we place long-running memcached instances as resident load. These instances serve read and write queries following the Facebook `etc` workload characteristics [23]. `etc` is the large memcached deployment in Facebook, has a 3:1 read:write ratio, and a value distribution between 1B and 1KB. Memcached occupies about 40% of the total system capacity and has a QoS target of 200usec for the 99th percentile of response latency.

The incoming jobs are homogeneous, short Spark tasks (100msec ideal duration, 20 tasks per job) that perform logistic regression. A total of 300k jobs are submitted over 900 seconds. Figure 7.13a shows the response times of the Spark tasks for the three schedulers. The centralized scheduler adds significant overheads, while Sparrow and Tarcil lead to small overheads and behave similarly for 80% of the tasks. For the remaining tasks, Sparrow increases response times significantly, as it is unaware of the interference induced by memcached. Tarcil maintains low response times for most tasks.

It is also important to consider the impact on the memcached load. Figure 7.13b shows the latency CDF of the memcached requests. The black diamond depicts the QoS constraint of 200usec for the 99th request percentile. With Tarcil and the centralized scheduler, memcached does not suffer as both schedulers attempt to minimize interference. Sparrow, however, leads to large latency increases for memcached. Even though the performance of the short tasks is satisfactory, not accounting for resource preferences has an impact on the longer jobs in the cluster. Finally, Figure 7.13c shows how the 99th percentile of memcached requests changes throughout the execution of the experiment. Initially memcached meets its QoS for all three schedulers. As the cluster becomes more loaded the tail latency increases significantly for Sparrow.

Note that a naïve coupling of Sparrow – for short jobs – with Quasar – for long jobs – is inadequate for three reasons. First, Tarcil achieves higher performance for short tasks because it accounts for their resource preferences. Second, even if the long-running resident load was scheduled using Quasar, scheduling short tasks with Sparrow would degrade its performance. Third, while the difference in execution time achieved by Quasar and Tarcil for long jobs is small, scheduling overheads are significantly reduced, without sacrificing the scheduling decision quality.

7.6.3 Large-Scale Evaluation

Methodology: We also evaluated Tarcil on a 400-server EC2 cluster with 10 server types ranging from 4 to 32 cores. The total core count in the cluster is 4,178. All servers are dedicated and managed only by the examined schedulers and there is no external interference from other workloads.

We use applications including short Spark tasks, longer Hadoop jobs, streaming Storm jobs [252], latency-critical services (memcached [170] and Cassandra [49]), and single-server benchmarks (SPEC CPU2006, PARSEC [40], etc.). In total, 7,200 workloads are submitted with 1 second inter-arrival times. These applications stress different resources, including CPU, memory and I/O (network, storage). We measure job performance (from submission to completion), cluster utilization, scheduling overheads and quality of allocation decisions.

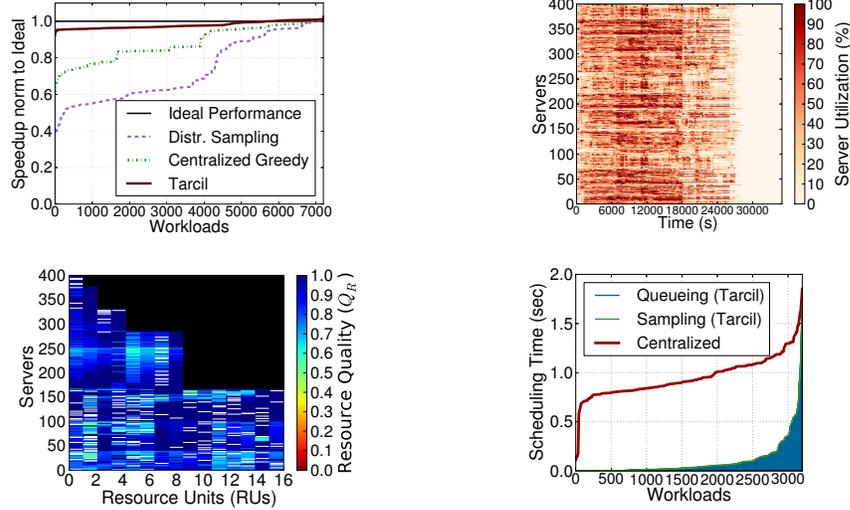


Figure 7.14: (a) Performance across 7,200 jobs on a 400-server EC2 cluster for the *Sampling-based* and *Centralized* schedulers and *Tarcil*, normalized to optimal performance, (b) cluster utilization achieved by *Tarcil* throughout the duration of the experiment, (c) quality of resource allocation across all RUs, and (d) scheduling overheads in *Tarcil* and the *Centralized* scheduler.

We compare *Tarcil*, *Quasar* and *Sparrow*. Because this scenario includes long-running jobs, such as *memcached*, that are not supported by the open-source implementation of *Sparrow*, we use *Sparrow* when applicable (e.g., *Spark*) and a *Sampling-based* scheduler that follows *Sparrow*'s principles (sample size 2, batch sampling and late binding) for the remaining jobs.

Performance: Figure 7.14a shows the performance (time between submission and completion) of the 7,200 workloads ordered from worst to best-performing, and normalized to their optimal performance in this cluster. Optimal corresponds to the performance on the best available resources and zero scheduling delay. The *Sampling-based* scheduler degrades performance for more than 75% of jobs. While *Centralized* behaves better, achieving an average of 82% of optimal, it still violates QoS for a large fraction of applications, particularly short-running workloads (0-3900 for this scheduler). *Tarcil* outperforms both schedulers, leading to 97% average performance and bounding maximum performance degradation to 8%.

Cluster utilization: Figure 7.14b shows the system utilization across the 400 servers of the cluster when incoming jobs are scheduled with *Tarcil*. CPU utilization is averaged across the cores of each server, and sampled every 2 sec. Utilization is

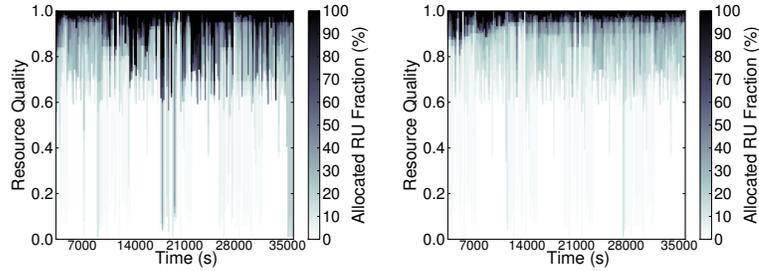


Figure 7.15: Resource quality CDFs for: (a) *Sampling-based*, (b) *Tarcil*.

70% on average at steady-state (middle of the scenario), when there are enough jobs to keep servers load-balanced. The maximum in the x-axis is set to the time it takes for the *Sampling-based* scheduler to complete the scenario ($\sim 35,000$ sec). The additional time corresponds to jobs that run on suboptimal resources and take longer to complete.

Core allocation: Figure 7.14c shows a snapshot of the RU quality across the cluster as observed by the job that is occupying each RU when using *Tarcil*. The snapshot is taken at 8,000s when all applications have arrived and the cluster operates at maximum utilization. White tiles correspond to unallocated resources. Dark blue tiles denote jobs with resources very close to their target quality. Lighter blue RUs correspond to jobs that received good but suboptimal resources. The graph shows that the majority of jobs are given appropriate resources. Note that high Q does not imply low server utilization. Utilization at the time of the snapshot is approximately 75%.

Scheduling overheads: Figure 7.14d shows the scheduling overheads for the Centralized scheduler and *Tarcil*. The results are consistent with the TPC-H experiment in Section 7.6.2. The overheads of the *Centralized* scheduler increase significantly with scale, adding approximately 1 sec to most workloads. *Tarcil* keeps overheads low, adding less than 150msec to more than 80% of workloads. This is essential for scalability. At high load, *Tarcil* increases the sample size to preserve the statistical guarantees and/or resorts to local queueing. The overheads for the *Sampling-based* scheduler are similar to *Tarcil* and are omitted from the graph for clarity.

Predictability: Figure 7.15 shows the fraction of allocated RUs that are over

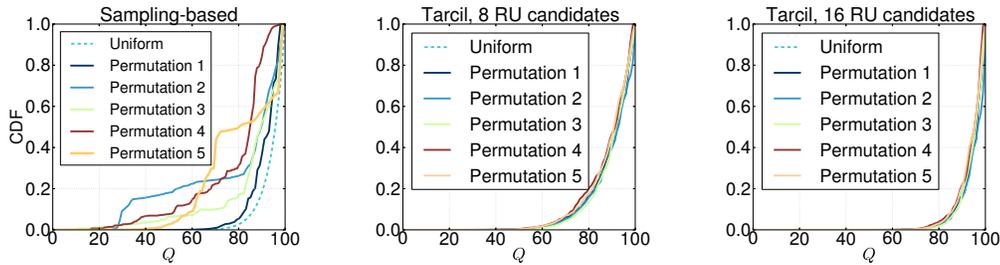


Figure 7.16: Resource quality distributions for the *Sampling-based* scheduler and *Tarcil* with $R = 8$ and 16 RUs across different permutations of the EC2 scenario.

a certain resource quality at each point of the duration of the scenario. Results are shown for the *Sampling-based* scheduler (left) and *Tarcil* (right). Darker colors towards the bottom of the graph denote that a larger fraction of allocated RUs have poor quality. At time 16,000sec, when the cluster is highly-loaded, the *Sampling-based* scheduler leads to 70% of allocated cores having quality less than 0.4. For *Tarcil*, only 18% of cores have less than 0.9 quality. Also note that, as the scenario progresses, the *Sampling-based* scheduler starts allocating resources of worse quality, while *Tarcil* maintains almost the same quality throughout the experiment.

Figure 7.16 explains this dissimilarity. It shows the CDF of resource quality for this scenario, and 5 random permutations of it (different job submission order). We show the CDF for the *Sampling-based* scheduler and *Tarcil* with 8 and 16 candidates. We omit the centralized scheduler which allocates resources of high quality most of the time. The sampling-based scheduler deviates significantly from the uniform distribution, since it does not account for the quality of allocated resources. In contrast, *Tarcil* closely follows the uniform distribution, improving the predictability of scheduling decisions.

7.7 Conclusions

We have presented *Tarcil*, a cluster scheduler that improves both scheduling speed and quality, making it appropriate for large, highly-loaded clusters running both short and long jobs. *Tarcil* uses an analytically-derived sampling framework that provides guarantees on the quality of allocated resources, and adjusts the sample size

to match application preferences. It also employs admission control to avoid excessive sampling and poor scheduling decisions at high load. We have compared Tarcil to existing parallel and centralized schedulers for a variety of workload scenarios on 100- to 400-server clusters on Amazon EC2. We have shown that it provides low scheduling overheads, high application performance, and high cluster utilization. Moreover, it reduces performance jitter, improving predictability in large, shared clusters.

Chapter 8

HCloud: Optimizing Resource Provisioning in Public Clouds

8.1 Introduction

An increasing amount of computing is now hosted in public clouds, such as Amazon's EC2 [13], Windows Azure [279] and Google Compute Engine [110], or in private clouds managed by frameworks such as VMware vCloud [268], OpenStack [206], and Mesos [139]. Cloud platforms provide two major advantages for end-users and cloud operators: *flexibility* and *cost efficiency* [29, 31, 136]. Users can quickly launch jobs without the overhead of setting up a new infrastructure every time. Cloud operators can achieve economies of scale by building large-scale datacenters (DCs) and by sharing their resources between multiple users and workloads.

Users can provision resources for their applications in two basic manners; using *reserved* and *on-demand* resources. *Reserved resources* consist of servers reserved for long periods of time (typically 1-3 years [13]) and offer consistent service, but come at a significant upfront cost for the purchase of the long-term resource contract. In the other extreme are *on-demand resources*, which can be full servers or smaller instances and are progressively obtained as they become necessary. In this case, the user pays only for resources used at each point in time, but the per hour cost is 2-3x higher compared to reserved resources. Moreover, acquiring on-demand resources induces

Configuration	Cost	Performance unpredictability	Spin-up	Flexibility	Typical usage
Reserved	High upfront, low per hour	no	no	no	long-term
On-demand	No upfront, high per hour	yes	yes	yes	short-term
Hybrid	Medium upfront, medium per hour	low	some	yes	long-term

Table 8.1: Comparison of system configurations with respect to: cost, performance unpredictability, overhead and flexibility.

instantiation overheads and depending on the type of instance, the variability in the quality of service obtained can be significant.

Since provisioning must determine the necessary resources, it is important to understand the extent of this unpredictability. Performance varies both across instances of the same type (spatial variability), and within a single instance over time (temporal variability) [28, 207, 148, 100, 160, 172, 185, 208, 221, 275, 231]. Figure 8.1 shows the variability in performance for a Hadoop job running a recommender system using Mahout [184] on various instance types on Amazon EC2 [13] and on Google Compute Engine (GCE) [110]. Analytics such as Hadoop and Spark [288] are throughput-bound applications, therefore performance here corresponds to the completion time of the job. The instances are ordered from smallest to largest, with respect to the number of virtual CPUs and memory allocations they provide. We show 1 vCPU `micro`, 1-8 vCPU standard (`stX`) and 16 vCPU memory-optimized instances (`mX`) [13, 110]. Each graph is the violin plot of completion time of the Hadoop job over 40 instances of the corresponding type. The dot shows the mean performance for each instance type. It becomes clear that especially for instances with less than 8 vCPUs unpredictability is significant, while for the `micro` instances in EC2 several jobs fail to complete due to the internal EC2 scheduler terminating the VM. For the larger instances (`m16`), performance is more predictable, primarily due to the fact that these instances typically occupy a large fraction of the server, hence they have a much lower probability of suffering from interference from co-scheduled workloads, excluding potential network interference. Between the two cloud providers, EC2 achieves higher average performance than GCE, but exhibits worse tail performance (higher unpredictability).

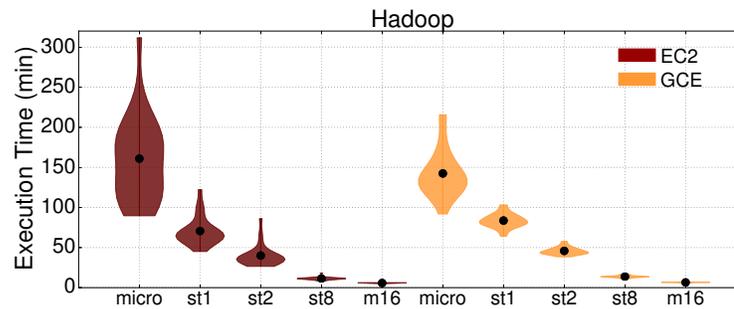


Figure 8.1: Performance unpredictability on Amazon EC2 and Google Compute Engine for a Hadoop job.

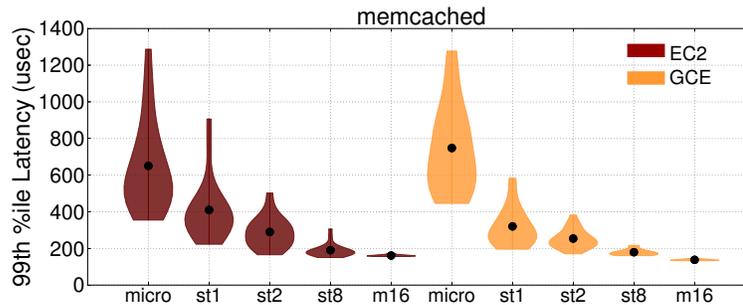


Figure 8.2: Performance unpredictability on Amazon EC2 and Google Compute Engine for memcached.

Figure 8.2 shows a similar experiment for a latency-critical service (memcached) on the same instance types. Note that the number of memcached clients is scaled by the number of vCPUs of each instance type, to ensure that all instances operate at a similar system load. Unpredictability is even more pronounced now, as memcached needs to satisfy tail latency guarantees [68], as opposed to average performance. The results from above hold, with the smaller instances (less than 8 vCPUs) experiencing significant variability in their tail latency. Performance jitter decreases again for the 8-16 vCPU VMs, especially in the case of the memory-optimized instances (m16). Additionally GCE now achieves better average and tail performance compared to EC2.

The goal of this work is to optimize *performance over cost for cloud systems*,

similarly to the way work on system design and resource management optimized performance per Watt for small- and large-scale systems [167, 297, 254, 166, 257]. We first explore the implications of the two main provisioning approaches (reserved and on-demand resources), with respect to performance variability and cost efficiency. We perform this analysis on Google Compute Engine (GCE) [110] using three representative workload scenarios with mixes of batch and latency-critical applications, and increasing levels of load variability. We assume no a priori knowledge of the applications in each scenario, except for the minimum and maximum aggregate load for each scenario, which is needed for a comparison with an idealized statically-reserved provisioning strategy.

Our study reveals that while reserved resources are superior with respect to performance (2.2x on average over on-demand), they require a long-term commitment, and are therefore beneficial for use cases over extended periods of time. Fully on-demand resources, on the other hand, are more cost-efficient for short-term use cases (2.5x on average), but are prone to performance unpredictability, especially when using smaller instances. They also incur instantiation overheads to spin-up new VMs. Our study also shows that to achieve reasonable performance predictability with either strategy, it is crucial to understand the resource preferences and sensitivity to interference of individual applications [79, 186, 223]. Recent work has shown that a combination of lightweight profiling and classification-based analysis can provide accurate estimations of job preferences with respect to the different instance types, the sensitivity to interference in shared resources and the amount of resources needed to satisfy each job's performance constraint (Chapter 4).

Next, we consider *hybrid provisioning strategies* that use both reserved (long-term) and on-demand (short-term) resources. A hybrid provisioning strategy has the potential to offer the best of both worlds by allowing users to leverage reserved resources for the steady-state long-term load, and on-demand resources for short-term resource needs. The main challenge with hybrid provisioning strategies is determining how to schedule jobs between the two types of resources. We show that leveraging the knowledge on resource preferences and accounting for the characteristics of on-demand resources, and the system load enables correct mapping of jobs to reserved

and on-demand resources. Table 8.1 shows the differences between the three main provisioning strategies with respect to *cost*, *performance unpredictability*, *instantiation overheads* and *provisioning flexibility*.

We demonstrate that hybrid provisioning strategies achieve both high resource efficiency and QoS-awareness. They maximize the usage of the already-provisioned reserved resources, while ensuring that applications that can tolerate some performance unpredictability will not delay the scheduling of interference-sensitive workloads. We also compare the performance, cost and provisioning needs of hybrid systems against the fully reserved and fully on-demand strategies examined before over a wide spectrum of workload scenarios. Hybrid provisioning strategies achieve within 8% of the performance of fully reserved systems (and 2.1x better than on-demand systems), while improving their cost efficiency by 46%. Reserved resources are utilized at 80% on average during steady-state. Finally, we perform a detailed sensitivity analysis of performance and cost with job parameters, such as duration, and system parameters such as resource pricing, spin-up overhead, and external load.

8.2 Cloud Workloads and Systems

8.2.1 Workload Scenarios

We examine the three workload scenarios shown in Figure 8.3 and summarized in Table 8.2. Each scenario consists of a mix of batch applications (Hadoop workloads running over Mahout [184] and Spark jobs) and latency-critical workloads (memcached). The batch jobs are machine learning and data mining applications, including recommender systems, support vector machines, matrix factorization, and linear regression. memcached is driven with loads that differ with respect to the read:write request ratio, the size of requests, the inter-arrival time distribution, the client fanout and the size of the dataset.

The first scenario has minimal load variability (*Static*). In steady-state the aggregate resource requirements are 854 cores on average. Approximately 55% of cores are required for batch jobs and the remaining 45% for the latency-critical services. The

difference between maximum and minimum load is 10% and most jobs last several minutes to a few tens of minutes.

Second, we examine a scenario with mild, long-term load variability (*Low Variability*). The steady-state minimum load requires on average 605 cores, while in the middle of the scenario the load increases to 900 cores. The surge is mostly caused by an increase in the load of the latency-critical applications. On average 55% of cores are needed for batch jobs and the remaining 45% for the latency-critical services.

Finally, we examine a scenario with large, short-term load changes (*High Variability*). The minimum load is 210 cores, while the maximum load reaches up to 1226 cores for short time periods. Approximately 60% of cores are needed for batch jobs and 40% for the latency-critical services. Because of the increased load variability, each job is shorter (8.1 min duration on average).

The ideal duration for each scenario, with no scheduling delays or degradation due to interference between workloads, is approximately 2 hours.

8.2.2 Cloud Instances

We use servers on Google Compute Engine (GCE) for all experiments. For provisioning strategies that require smaller instances we start with the largest instances (16 vCPUs) and partition them using Linux containers [26, 66]. The reason for constructing smaller instances as server slices as opposed to directly requesting various instance types is to introduce controlled external interference which corresponds to typical load patterns seen in cloud environments, rather than the random interference patterns present at the specific time we ran each experiment. This ensures *repeatable experiments* and *consistent comparisons* between provisioning strategies.

We model external interference by imposing external load that fluctuates $\pm 10\%$ around a 25% average utilization [31, 79]. The external load is generated using both batch and latency-critical workloads. Section 8.5.1 includes a sensitivity study to the intensity of external load.

We only partition servers at the granularity of existing GCE instances, e.g., 1,2,4,8 and 16 vCPUs. Whenever we refer to the cost of an on-demand instance, we quote

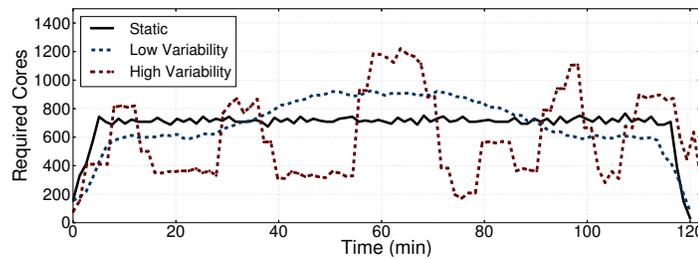


Figure 8.3: The three workload scenarios.

the cost of the instance that would be used in the real environment, e.g., a 2 vCPU instance. Similarly, we account for the spin-up overhead of the instance of the desired size, wherever applicable. Finally, all scheduling actions such as autoscale and migration performed by GCE are disabled.

8.2.3 Cloud Pricing

Google Compute Engine currently only offers on-demand instances. To encourage high instance usage, it provides sustained usage monthly discounts [110]. Although sustained usage discounts reduce the prices of on-demand instances, they do not approximate the price of long-term reserved resources. The most popular alternative pricing model is the one used by AWS, which includes both long-term resource reservations and short-term on-demand instances. Because this pricing model offers more provisioning flexibility and captures the largest fraction of the cloud market today, we use it to evaluate the different provisioning strategies and adapt it to the resource prices of GCE. Specifically, we approximate the cost of reserved resources on GCE based on the reserved to on-demand price ratio for EC2, adjusted to the instance prices of GCE. In Section 8.5.3 we discuss how our results translate to different pricing models, such as the default GCE model and the pricing model used by Windows Azure.

	Workload Scenarios		
	Static	Low Var	High Var
max:min resources ratio	1.1x	1.5x	6.2x
batch:low-latency – in jobs	4.2x	3.6x	4.1x
– in cores	1.4x	1.4x	1.5x
inter-arrival times (sec)	1.0	1.0	1.0
ideal completion time (hr)	2.1	2.0	2.0

Table 8.2: Workload scenario characteristics.

8.3 Provisioning Strategies

The two main types of resource offerings in cloud systems are *reserved* and *on-demand* resources. Reserved instances require a high upfront capital investment, but have 2-3x lower per-hour cost than on-demand resources, offer better service availability (1-year minimum), and provide consistent performance. On-demand resources are charged in a pay-as-you-go manner, but incur spin-up overheads and experience performance unpredictability due to interference from external load. We ignore spot instances for the purpose of this work, since they do not provide any availability guarantees.

The provisioning strategy must acquire the right type and number of resources for a workload scenario. Ideally, a provisioning strategy achieves three goals: (1) *high workload performance*, (2) *high resource utilization (minimal overprovisioning)*, and (3) *minimal provisioning and scheduling overheads*. We initially study the three obvious provisioning strategies described in Table 8.3: a statically-provisioned strategy using only reserved resources (SR); an on-demand strategy (OdF) that only uses full servers (16 vCPU instances); and an on-demand strategy (OdM) that uses instances of any size and type.

8.3.1 Statically Reserved Resources (SR)

This strategy statically provisions reserved resources for a 1 year period, the shortest contract for reserved resources on cloud systems such as EC2. Reserved resources require significant capital investment upfront, although the *per-hour* charge is 2-3x lower than for the corresponding on-demand instances. Moreover, reserved resources

are readily available as jobs arrive, eliminating the overhead of spinning up new VMs on-demand. Because SR only reserves large instances (16 vCPU), there is limited interference from external load, except potentially for some network interference.

Because of its static nature, SR must provision resources for the peak requirements of each workload scenario, plus a small amount of overprovisioning. Overprovisioning is needed because all scenarios contain latency-critical jobs, that experience tail latency spikes when using nearly saturated resources [29, 31, 68, 162]. We explain the insight behind the amount of overprovisioning in Section 8.3.3. Peak requirements can be easily estimated for mostly static workload scenarios. For scenarios with load variability, static provisioning results in acquiring a large number of resources which remain underutilized for significant periods of time.

8.3.2 Dynamic On-Demand Resources (OdF, OdM)

We now examine two provisioning strategies that acquire resources as they become necessary to accommodate the incoming jobs of each workload scenario. In this case there is no need for a large expenditure upfront, but the price of each instance per hour is 2-3x higher compared to the corresponding reserved resources. Moreover, each new instance now incurs the overhead needed to spin up the new VMs. This is typically 12-19 seconds for GCE, although the 95th percentile of the spin-up overhead is up to 2 minutes. Smaller instances tend to incur higher spin-up overheads.

Because of spin-up overheads, these two strategies must also decide how long they should retain the resources for after a job completes. If, for example, a workload scenario has no or little load variability, instances should be retained to amortize the spin-up overhead. On the other hand, retaining instances when load variability is high can result in underutilized resources. We determine retention time by drawing from related work on processor power management. The challenge in that case is to determine when to switch to low power modes that enable power savings but incur overheads to revert to an active mode [242, 180, 189]. Given that the job inter-arrival time in our scenarios is 1 second, we set the retention time to 10x the spin-up overhead

	SR	OdF	OdM	HF	HM
Reserved resources	Yes	No	No	Yes	Yes
On-demand resources	No	Yes (full servers)	Yes	Yes (full servers)	Yes

Table 8.3: Resource provisioning strategies.

of an instance.¹ Section 8.5.1 shows a sensitivity analysis to retention time. Only instances that perform in a satisfactory manner are retained past the completion of their jobs.

We examine two variants of on-demand provisioning strategies. On-demand Full (OdF) only uses large instances (16 vCPUs), which are much less prone to external interference (see Section 8.1). On-demand Mixed (OdM) acquires on-demand resources of any instance type, including smaller instances with 1-8 vCPUs. While OdM offers more flexibility, it introduces the issue that performance unpredictability due to external interference now becomes substantial. There are ways to improve performance predictability in fully on-demand provisioning strategies, e.g., by sampling multiple instances for each required instance and only keeping the better-behaved instances [100]. Although this approach addresses the performance variability across instances, it is still prone to temporal variation within a single instance. Additionally, it is only beneficial for long-running jobs that can afford the overhead of sampling multiple instances. Short jobs, such as real-time analytics (100msec-10sec) cannot tolerate long scheduling delays and must rely on the initial resource assignment.

8.3.3 The Importance of Resource Preferences

So far, we have assumed that the provisioning strategy has limited knowledge about the resource preferences of individual jobs within a workload scenario. Traditionally, the end-users have to specify how many resources each job should use; unfortunately this is known to be error-prone and to frequently lead to significant resource over-provisioning [31, 223, 177, 79, 48]. Moreover, this offers no insight on the sensitivity

¹The benefit of longer retention time varies across instance sizes due to differences in spin-up overheads.

of each job to interference from other jobs, external or not, running on the same physical server. This is suboptimal for both the statically-reserved and on-demand strategies, which will acquire more/less resources than what is truly needed by an application. The lack of interference understanding is equally problematic. SR will likely colocate jobs that interfere negatively with each other on the same instance. OdF and OdM will likely acquire instance types that are prone to higher interference than what certain jobs can tolerate.

The recently-proposed Quasar system provides a methodology to quickly determine the resource preferences of new jobs [79]. When a job is submitted to the system, it is first profiled on two instance types, while injecting interference in two shared resources, e.g., last level cache and network bandwidth. This profiling signal is used by a set of classification techniques which find similarities between the new and previously-scheduled jobs with respect to instance type preferences and sensitivity to interference. A job's sensitivity to interference in resource i is denoted by c_i , where $i \in [1, N]$, and $N = 10$ the number of examined resources [79]. Large values of c_i mean that the job puts a lot of pressure in resource i . To capture the fact that certain jobs are more sensitive to specific resources we rearrange vector $C = [c_1, c_2, \dots, c_N]$ by order of decreasing magnitude of c_i , $C' = [c_j, c_k, \dots, c_n]$. Finally, to obtain a single value for C' , we use an order preserving encoding scheme as follows: $Q = c_j \cdot 10^{(2 \cdot (N-1))} + c_k \cdot 10^{(2 \cdot (N-2))} + \dots + c_n$, and normalize Q in $[0, 1]$. Q denotes the resource quality a job needs to satisfy its QoS constraints. High Q denotes a resource-demanding job, while low Q a job that can tolerate some interference in shared resources.

We use Quasar's estimations of resource preferences and interference sensitivity to improve resource provisioning. For SR, we use these estimations to find the most suitable resources available in the reserved instances with respect to resource size and interference using a simple greedy search [79]. Accounting for the information on resource preferences reduces overprovisioning to 10-15%. For OdF, the estimations are used to select the minimum amount of resources for a job, and to match the resource capabilities of instances to the interference requirements of a job. For OdM, this additionally involves requesting an appropriate instance size and type (standard,

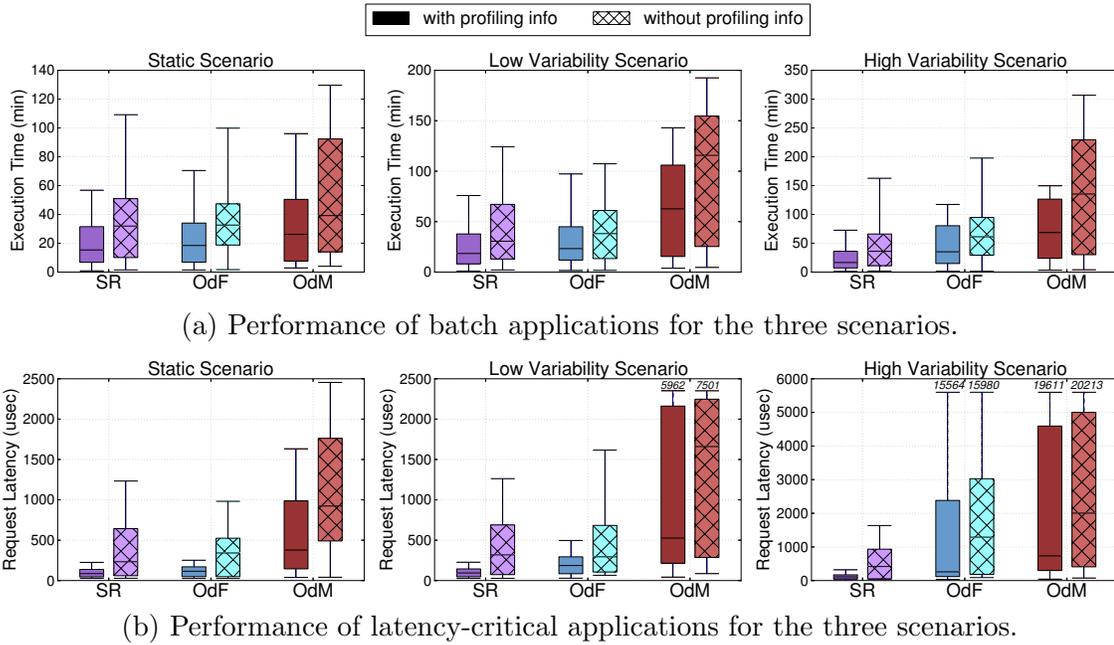


Figure 8.4: Performance of jobs of the three workload scenarios with the three provisioning strategies. The boundaries of the boxplots depict the 25th and 75th percentiles, the whiskers the 5th and 95th percentiles and the horizontal line in each boxplot shows the mean.

compute- or memory-optimized). Note that because smaller instances are prone to external interference, provisioning decisions may have lower accuracy in this case.

Finally, we must detect suboptimal application performance and revisit the allocation decisions at runtime [224, 24, 25, 54, 79]. Once an application is scheduled its performance is monitored and compared against its expected QoS. If performance drops below QoS we take action [79]. At a high level, we first try to restore performance through local actions, e.g., increasing the resource allocation, and then through rescheduling. Rescheduling is very unlikely in practice.

8.3.4 Provisioning Strategies Comparison

Performance: We first compare the performance impact of the three provisioning strategies, with and without Quasar’s information on individual job preferences.

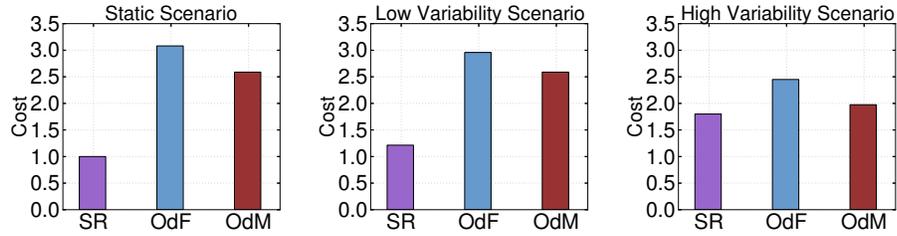


Figure 8.5: Cost of fully reserved and on-demand systems.

Figure 8.4 shows the performance achieved by each of the three provisioning strategies for the three workload scenarios. We separate batch (Hadoop, Spark) from latency-critical applications (memcached), since their critical performance metric is different: completion time for the batch jobs and request latency distribution for memcached. The boundaries in each boxplot depict the 25th and 75th percentiles of performance, the whiskers the 5th and 95th percentile and the horizontal line shows the mean. When the information from Quasar is not used, the resources for each job are sized based on user-defined resource reservations. For batch jobs (Hadoop and Spark) this translates to using the default framework parameters (e.g., 64KB block size, 1GB heapsize for Hadoop), while for memcached resources are provisioned for peak load [223]. OdM requests the smallest instance size that satisfies the resource demands of a job. SR allocates resources for workloads on the reserved instances with the most available resources (least-loaded).

It is clear from Figure 8.4 that all three provisioning strategies benefit significantly from understanding the jobs' resource preferences and interference sensitivity. Specifically for SR, there is a 2.4x difference in performance on average across scenarios. The differences are even more pronounced in the case of latency-critical applications, where the performance metric of interest is tail, instead of average performance. Omitting the information on interference sensitivity in this case significantly hurts request latency. In all following results, we assume that provisioning takes job preferences into account, unless otherwise stated.

We now compare the performance achieved by the three provisioning strategies. The static strategy SR achieves the best performance for all three scenarios, both for batch and latency-critical workloads. OdF behaves near-optimally for the static

scenario, but worsens for the scenarios where variability is present. The main reason is the spin-up overhead required to obtain new resources as they become necessary. Strategy OdM achieves the worst performance of all three provisioning strategies for every scenario (2.2x worse than SR on average), in part because of the spin-up overhead, but primarily because of the performance unpredictability it experiences from external load in the smaller instances. Memcached suffers a 24x and 42x increase in tail latency in the low- and high-variability scenarios, as it is more sensitive to resource interference.

Cost: Figure 8.5 shows the relative cost of each strategy for the three scenarios. All costs are normalized to the cost of the static scenario with SR. Although strategy SR appears to have the lowest cost for a 2 hour run (2-3x lower per hour charge than on-demand), it requires at least a 1-year commitment with all charges happening in advance. Therefore, unless a user plans to leverage the cluster for long periods of time, on-demand resources are dramatically more cost-efficient. Moreover, SR is not particularly cost effective in the presence of high workload variability, since it results in significant overprovisioning. Between the two on-demand strategies, OdM incurs lower cost, since it uses smaller instances, while OdF only uses the largest instances available. Note however that the cost savings of OdM translate to a significant performance degradation due to resource unpredictability (Figure 8.4).

8.4 Hybrid Provisioning Strategies

The previous section showed that neither fully reserved nor fully on-demand strategies are ideal. Hybrid provisioning strategies that combine reserved and on-demand resources have the potential to achieve the best of both worlds. This section presents two hybrid provisioning strategies that intelligently assign jobs between reserved and on-demand resources and compares their performance and cost against the strategies of Section 8.3. Again, we make use of the information on resource preferences and interference sensitivity of individual jobs, as estimated by Quasar.

8.4.1 Provisioning Strategies

We design two hybrid strategies that use both reserved and on-demand resources. The first strategy (HF) only uses large instances for the on-demand resources, to reduce performance unpredictability. The second strategy (HM), uses a mix of on-demand instance types to reduce cost, including smaller instances that experience interference from external load. The retention time policy of on-demand resources is the same as for the purely on-demand strategies OdF and OdM. The reserved resources in both cases are large instances, as with the statically-provisioned strategy (SR). We configure the number of reserved instances to accommodate the minimum steady-state load, e.g., 600 cores for the low variability scenario to avoid overprovisioning of reserved resources. For scenarios with low steady-state load but high load variability the majority of resources will be on-demand.

Since HF uses large instances with limited performance unpredictability for both reserved and on-demand resources, it mostly uses on-demand instances to serve overflow load. In contrast, with HM on-demand instances may be smaller and can experience resource interference from external load. Therefore, for hybrid strategies it is critical to determine which jobs should be mapped to reserved versus on-demand resources, based on their interference sensitivity and the availability of reserved resources.

8.4.2 Application Mapping Policies

We first consider a baseline policy that maps applications between the reserved and on-demand resources randomly using a fair coin. Figure 8.6 shows the performance of applications mapped to the reserved (left) and on-demand resources (right) for the two hybrid provisioning strategies in the case of the high variability scenario. Performance is normalized to the performance each job achieves if it runs with unlimited resources alone in the system (in isolation). Figure 8.7 also shows the utilization of the reserved instances and the total cost to run the 2 hour scenario normalized to the cost of the static scenario with SR. Because of the large number of scheduled applications, approximately half of them will be scheduled on reserved and half on on-demand

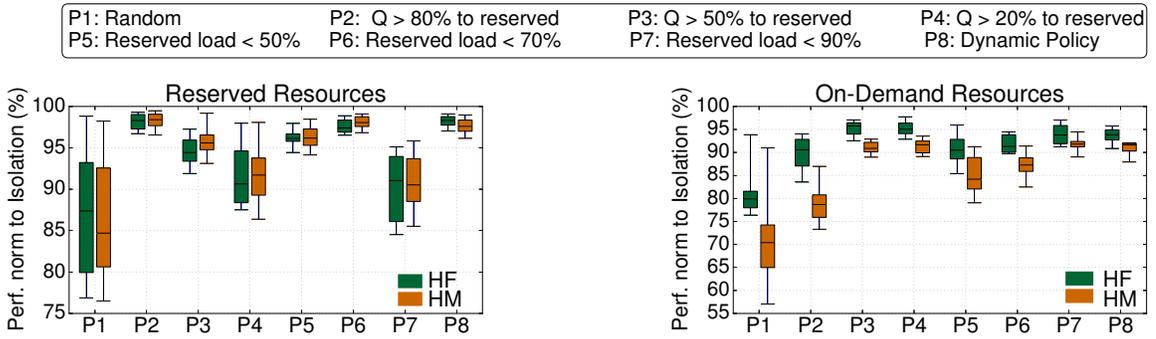


Figure 8.6: Sensitivity to the policy of mapping jobs to reserved versus on-demand resources for HF and HM.

resources [123]. The random policy hurts performance for jobs mapped to either type of resources. In the reserved resources, performance degrades as more workloads than the instances can accommodate are assigned to them, and are therefore queued. In the on-demand resources, performance degrades for two reasons. First, because of the inherent unpredictability of resources, especially in the case of HM, and, more prominently, because jobs that are sensitive to interference and should have been mapped to reserved resources slow down due to external load.

Ideally, the mapping policy should take into account the sensitivity of jobs to performance unpredictability. The following three policies shown in Figure 8.6 set a limit to the jobs that should be mapped to reserved resources based on the quality of resources they need. $P2$ assigns jobs that need quality $Q > 80\%$ to the reserved instances to protect them from the variability of on-demand resources. $P3$ and $P4$ set stricter limits, with $P4$ only assigning very tolerant to unpredictability jobs to the on-demand resources. As we move from $P2$ to $P4$ the performance of jobs in the on-demand instances improves, as the number of applications mapped to them decreases. In contrast, the performance of jobs scheduled to reserved resources worsens due to increased demand and queueing for resources. In general, performance is worse for HM in the on-demand resources, due to the increased performance variability of smaller instances.

It is clear that there needs to be an upper load limit for the reserved resources. The next three policies $P5 - P7$ set progressively higher, static limits. For low utilization

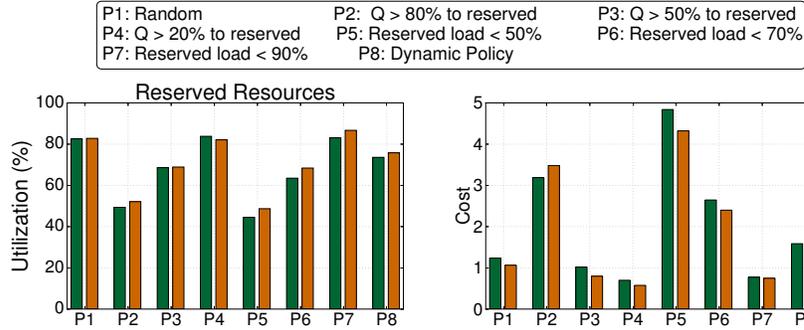


Figure 8.7: CPU utilization of reserved resources and cost with different application mapping policies for HF and HM.

limits, e.g., 50-70% the performance of jobs on reserved resources is near-optimal. In contrast, jobs assigned to on-demand resources suffer substantial performance degradations, since application mapping is only determined based on load and not based on resource preferences. For a utilization limit of 90%, the performance of jobs in the reserved resources degrades due to excessive load. Low utilization in the reserved resources also significantly increases the cost, as additional on-demand resources have to be obtained. Therefore a policy using a static utilization limit that does not distinguish between the resource preferences of jobs is also suboptimal.

Based on these findings we design a dynamic policy to separate jobs between reserved and on-demand resources. The policy adheres to three principles. First, it utilizes reserved resources before resorting to on-demand resources. Second, applications that can be accommodated by on-demand resources should not delay the scheduling of jobs sensitive to resource quality. Third, the system must adjust the utilization limits of reserved instances to respond to performance degradations due to excessive queueing.

Figure 8.8 explains the dynamic policy. We set two utilization limits for the reserved resources. First, a *soft limit* is set (experimentally set at 60-65% utilization), below which all incoming jobs are allocated reserved resources. Once utilization exceeds this limit, the policy differentiates between applications that are sensitive to performance unpredictability and applications that are not. The differentiation is done based on the resource quality Q a job needs to satisfy its QoS constraints and

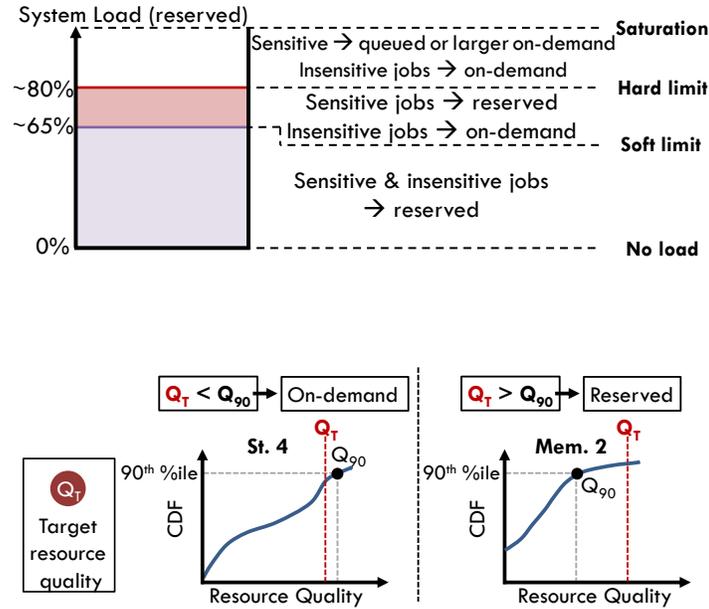


Figure 8.8: Application mapping scheme between reserved and on-demand instances for HF and HM. Figure 8.8a shows the resource limits that determine where applications are scheduled, and Figure 8.8b shows how an application is scheduled to on-demand versus reserved resources based on its performance constraints.

the knowledge on the quality of previously-obtained on-demand instances. Once we determine the instance size a job needs (number of cores, memory and storage), we compare the 90th percentile of quality of that instance type (monitored over time) against the target quality (Q_T) the job needs. If $Q_{90} > Q_T$ the job is scheduled on the on-demand instance, otherwise it is scheduled on the reserved instances. Examining the 90th percentile is sufficient to ensure accurate decisions for the majority of jobs.

Second, we set a *hard limit* for utilization, when jobs need to get queued until reserved resources become available. At this point, any jobs for which on-demand resources are satisfactory are scheduled in the on-demand instances and all remaining jobs are locally queued [209]. An exception occurs for jobs whose queueing time is expected to exceed the time it would take to spin up a large on-demand instance (16 vCPUs); these jobs are instead assigned to on-demand instances. Queueing time is estimated using a simple feedback loop based on the rate at which instances of a

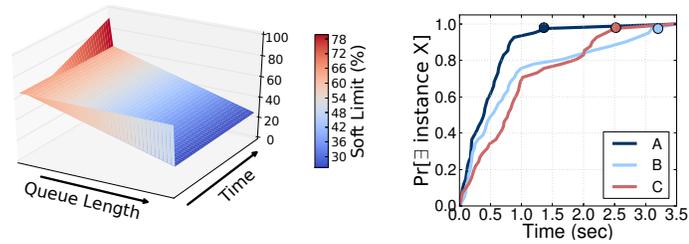


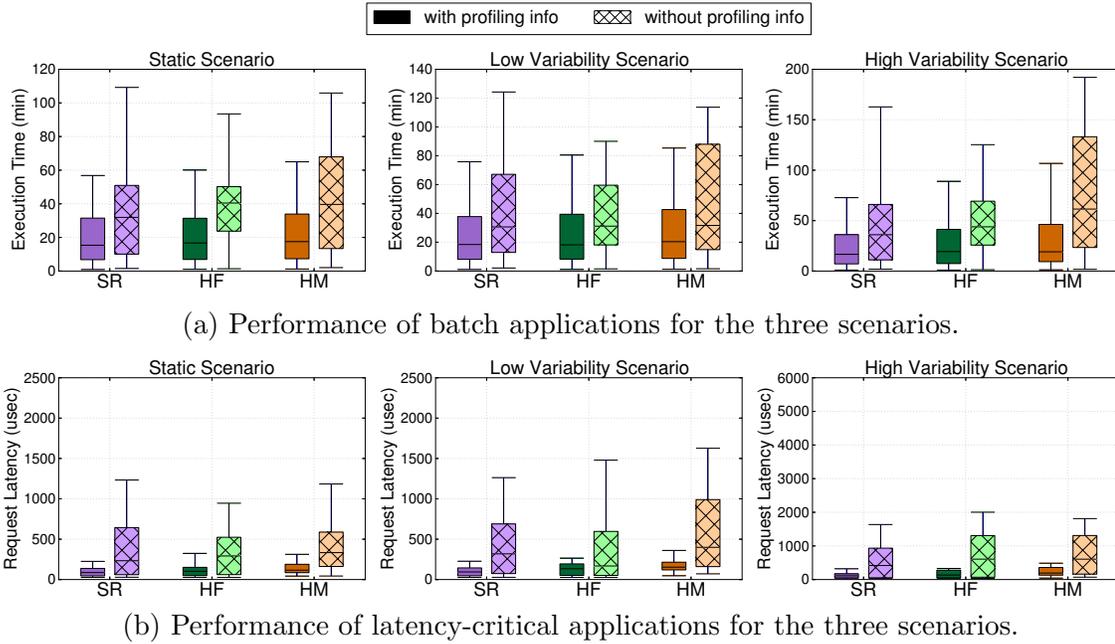
Figure 8.9: Determining the soft utilization limit (left) and the expected waiting time (right) in HF and HM.

given type are being released over time. For example, if out of 100 jobs waiting for an instance with 4 vCPUs and 15GB of RAM, 99 were scheduled in less than 1.4 seconds, the system will estimate that there is a 0.99 probability that the queuing time for a job waiting for a 4 vCPU instance will be 1.4 seconds. Figure 8.9b shows a validation of the estimation of waiting time for three instance types. The lines show the cumulative distribution function (CDF) of the probability that an instance of a given type becomes available. The dots show the estimated queuing time for jobs waiting to be scheduled on instances with 4 (A), 8 (B) and 16 vCPUs (C) in the high variability scenario. In all cases the deviation between estimated and measured queuing time is minimal.

Third, we adjust the soft utilization limit based on the rate at which applications get queued. If the number of queued jobs increases sharply, the reserved instances should become more selective in the workloads they accept, i.e., the soft limit should decrease. Similarly, if no jobs get queued for significant periods of time, the soft limit should increase to accept more incoming jobs. We use a simple feedback loop with linear transfer functions to adjust the soft utilization limit of the reserved instances as a workload scenario progresses. Figure 8.9a shows how the soft limit changes with execution time and queue length.

8.4.3 Provisioning Strategies Comparison

Performance: Figure 8.10 compares the performance achieved by the static strategy SR and the two hybrid strategies (HF and HM), with and without the profiling



(a) Performance of batch applications for the three scenarios.

(b) Performance of latency-critical applications for the three scenarios.

Figure 8.10: Performance of the three scenarios with the statically-reserved and hybrid provisioning strategies. The boundaries of the boxplots depict the 25th and 75th percentiles, the whiskers the 5th and 95th percentiles and the horizontal line in each boxplot shows the mean.

information for new jobs. Again we separate batch from latency-critical jobs. As expected, having the profiling information improves performance significantly for the hybrid strategies, for the additional reason that it is needed to decide which jobs should be scheduled on the reserved resources (2.4x improvement on average for HF and 2.77x for HM). When using the profiling information, strategies HF and HM come within 8% of the performance of the statically reserved system (SR), and in most cases outperform strategies OdF and OdM, especially for the scenarios with significant load variability. The main reason why HF and HM achieve good performance is that they differentiate between applications that can tolerate the unpredictability of on-demand instances, and jobs that need the predictable performance of a fully controlled environment. Additionally hybrid strategies hide some of the spin-up overhead of on-demand resources by accommodating part of the load in the reserved instances.

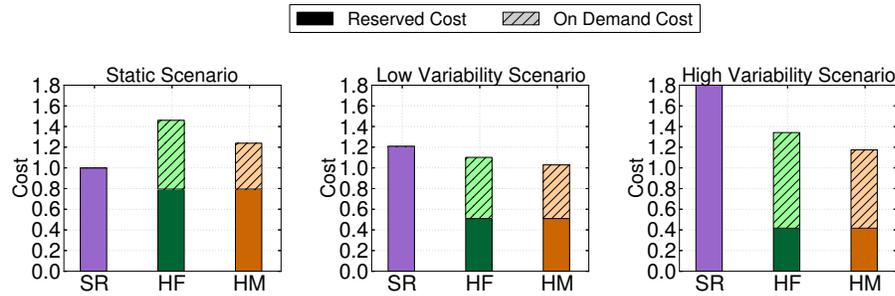


Figure 8.11: Cost comparison between SR, HF and HM.

Cost: Figure 8.11 shows the relative cost of strategies SR, HF and HM for the three scenarios. While the static provisioning strategy (SR) is more cost-efficient in the static scenario where provisioning is straight-forward, the hybrid strategies incur significantly lower costs for both scenarios with load variability. Therefore, unless load is almost completely static, statically-provisioned resources is not cost-efficient both due to long-term reservations, and significant overprovisioning. Additionally, because of the lower per-hour cost of reserved resources in HF and HM, the hybrid strategies have lower per-hour cost than fully on-demand resources as well. For HF and HM, most of the cost per hour comes from on-demand resources, since reserved instances are provisioned for the minimum steady-state load. Finally, between the two hybrid strategies, HM achieves higher cost-efficiency since it uses smaller instances.

8.5 Discussion

8.5.1 Sensitivity to Job/System Parameters

We first evaluate the sensitivity of the previous findings to various system and workload parameters. Unless otherwise specified, we use the same strategies as before to provision reserved and/or on-demand resources.

Resource cost: The current average cost ratio of on-demand to reserved resource per hour is 2.74. Figure 8.12 shows how the relative cost of the three scenarios varies for each of the five strategies when this ratio changes. The current ratio is shown with a vertical line at 2.74. All costs are normalized to the cost for the static scenario



Figure 8.12: Sensitivity to on-demand:reserved cost.

using SR. We change the ratio by scaling the price of reserved resources. We vary the ratio in $[0.01, 4]$; beyond that point the cost of SR per hour becomes practically negligible. Initially (0.01), strategies using only on-demand resources (OdF, OdM) are significantly more cost-efficient, especially for the scenarios with load variability. For the static scenario, even when on-demand resources are much cheaper than reserved, SR, HF and HM incur similar charges as the fully on-demand systems. For each scenario, there is a price ratio for which SR becomes the most cost-efficient strategy. As variability increases, this value becomes larger (e.g., for the high variability scenario the ratio needs to become 3 for SR to be more cost-efficient per hour than HM). Note that SR still requires at least a 1-year commitment, in contrast to the on-demand strategies. Finally, the hybrid strategies achieve the lowest per-hour cost for significant ranges of the price ratio, especially for scenarios with load variability.

Scenario duration: Figure 8.13 shows how cost changes for each strategy, as the scenario duration increases. Because we compare aggregate costs (instead of per-hour), this figure shows the absolute cost in dollars for each strategy. For the static scenario, from a cost perspective, strategy HM is optimal only if duration is $[20 - 25]$ weeks. For durations less than 20 weeks, strategy OdM is the most cost-efficient, while for durations more than 25 weeks the statically-reserved system (SR) is optimal. This changes for scenarios with load variability. Especially in the case of high variability, for durations larger than 18 weeks, strategy HM is the most cost-efficient, with the significantly overprovisioned reserved system (SR) never being the most efficient. Note that the charge for SR doubles beyond the 1 year (52 weeks) mark.

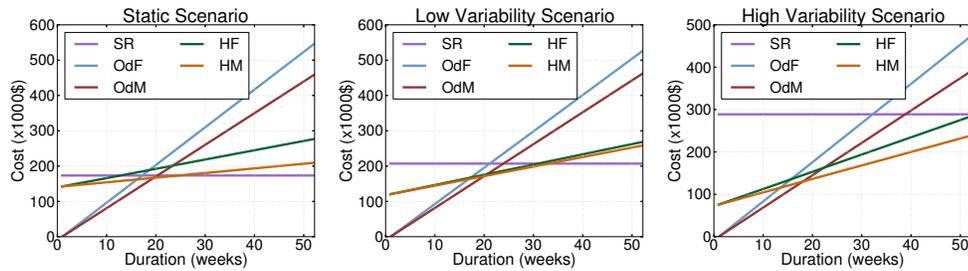


Figure 8.13: Sensitivity to scenario duration.

Spin-up overhead: Figure 8.14a shows how the 95th percentile of performance changes as the overhead to spin-up new resources changes for the high variability scenario. The statically-reserved strategy (SR) is obviously not affected by this change. Because in this scenario resources are frequently recycled, increasing the spin-up overhead significantly affects performance. This is more pronounced for the strategies using exclusively on-demand resources (OdF, OdM). The additional degradation for OdM comes from the performance unpredictability of smaller on-demand instances.

External load: Figure 8.14b shows the sensitivity of performance to external load (load in machines due to jobs beyond those provisioned with our strategies). SR provisions a fully-controlled environment, therefore there is no external load to affect performance. OdF and HF are also tolerant to external load, as they only use the largest instances, which are much less prone to external interference. For HM performance degrades minimally until 50% load, beyond which point the estimations on resource quality become inaccurate. OdM suffers most of the performance degradation as all of its resources are susceptible to external interference.

Retention time: Figure 8.15 shows the 95th percentile of performance and the cost of each strategy, as the time for which idle instances are maintained changes for the high variability scenario. As expected, releasing well-behaved instances immediately hurts performance, as it increases the overheads from spinning-up new resources. This is especially the case for this scenario, where load changes frequently. With respect to cost, higher retention time increases the cost of strategies using only on-demand resources (OdF, OdM), while SR remains unchanged; the difference for hybrid strategies is small. An unexpected finding is that excessive resource retention slightly

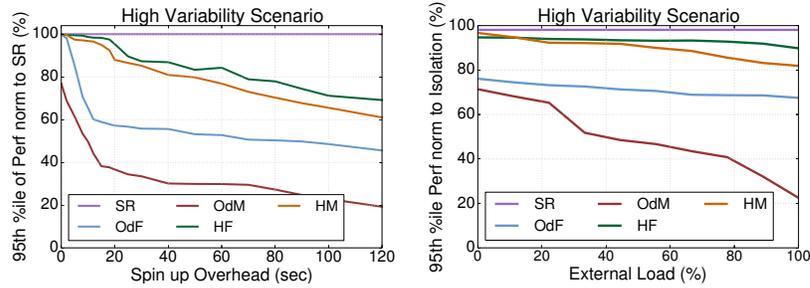


Figure 8.14: Sensitivity to spin-up time and external load.

hurts performance for OdM and HM. The primary reason is the temporal variability in the quality of on-demand resources, which degraded by the time new applications were assigned to these instances.

8.5.2 Provisioning Overheads

In the presented strategies, the provisioning overheads include job profiling and classification (Quasar), provisioning decisions, spin-up of new on-demand instances (where applicable), and rescheduling actions. The profiling that generates the input signal for the classification engine takes 5-10 sec, but only needs to happen the first time a job is submitted. Classification itself takes 50msec on average. Decision overheads include the greedy scheduler in the statically-reserved strategy (SR) and the overhead of deciding whether to schedule a new job on reserved versus on-demand resources in the hybrid strategies. In all cases decision overheads do not exceed 20msec, three orders of magnitude lower than the spin-up overheads of on-demand instances (10-20sec on average). Finally, job rescheduling due to suboptimal performance is very infrequent for all strategies except OdM, where it induces 6.1% overheads to the execution time of jobs on average.

8.5.3 Different Pricing Models

So far we have assumed a pricing model for reserved and on-demand instances similar to the one used by Amazon's AWS. This is a popular approach followed by many smaller cloud providers. Nevertheless, there are alternative approaches. GCE does

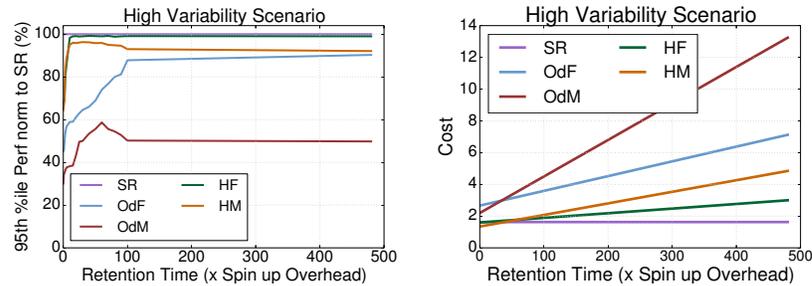


Figure 8.15: Sensitivity to resource retention time.

not offer long-term reservations. Instead it provides *sustained usage monthly discounts* to encourage high-utilization of on-demand resources. The higher the usage of a set of instances of a type for a fraction of the month, the lower the per-hour instance price for the remainder of the month. This approach does not differentiate whether one uses a single instance of type A for 3 weeks or 3 instances of type A for 1 week each. Microsoft Azure only offers on-demand resources of different types.

Even without reserved resources, the problem of selecting the appropriate instance size and configuration, and determining how long to keep an instance before releasing it remains. Figure 8.16 shows how cost changes for the three workload scenarios, under the Azure (on-demand only) and GCE (on-demand + usage discounts) pricing models, compared to the AWS pricing model (reserved + on-demand). We assume that the resources will be used at least for a one month period, so that GCE discounts can take effect. Cost is normalized to the cost of the static workload scenario under the SR provisioning strategy using the *reserved & on-demand* pricing model. Even with these alternative pricing models using the hybrid strategies and accounting for the resource preferences of incoming applications to optimize provisioning significantly benefits cost. For example, for the high variability scenario HM achieves 32% lower cost than OdF with the Windows Azure pricing model; similarly for the GCE model with discounts, HM achieves 30% lower cost than OdF.

GCE decouples the level of usage from the specific instance used. For example, monthly usage is considered the same between a single instance used for 50% of the month, and N instances of the same type used for $(50/N)\%$ of the month each. This introduces new opportunities to optimize resource provisioning by maximizing the

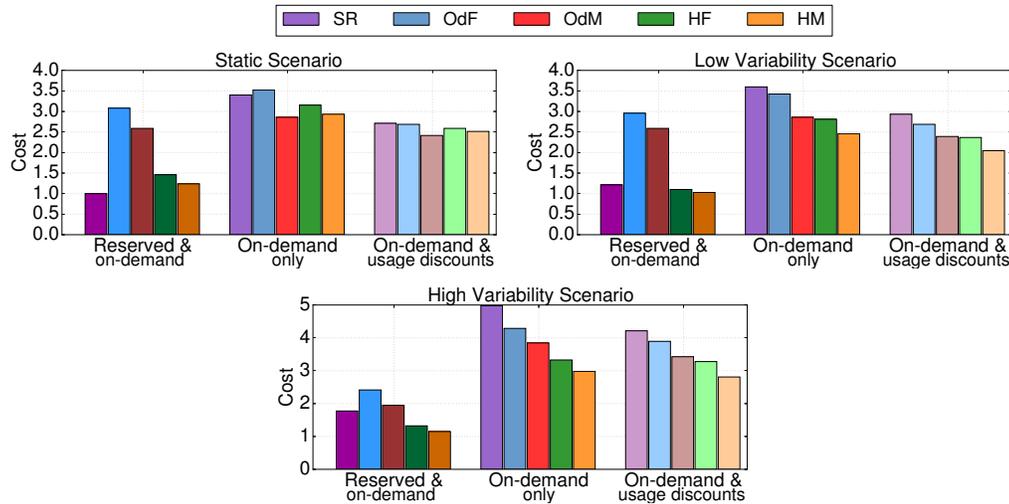


Figure 8.16: Sensitivity to the cloud pricing model for the three workload scenarios.

time a certain instance type is used during a month. We defer such considerations to future work.

8.5.4 Resource Efficiency

Apart from lowering cost, we also want to ensure that a provisioning strategy is not wasteful in terms of resources. Figure 8.17 shows the resource allocation by each strategy throughout the duration of the high variability scenario. The reserved system (SR) is provisioned statically for the peak requirements plus a 15% overprovisioning as described in Section 8.3.1. Because all instances are private (limited external interference) and all resources are readily available, the scenario achieves near-ideal execution time (~ 2 hr). However, because there is high load variability, utilization is rarely high, resulting in poor resource efficiency. Strategy OdF obtains resources as they become necessary and because it induces spin-up overheads frequently due to the constant load change, it results in longer execution time (132 min). It also introduces some overprovisioning, as it only requests the largest instances to constrain performance unpredictability. OdM does not overprovision allocations noticeably since it uses smaller instances, however, it significantly hurts performance, resulting in the scenario completing in 48% more time. Performance degradation is partially the result

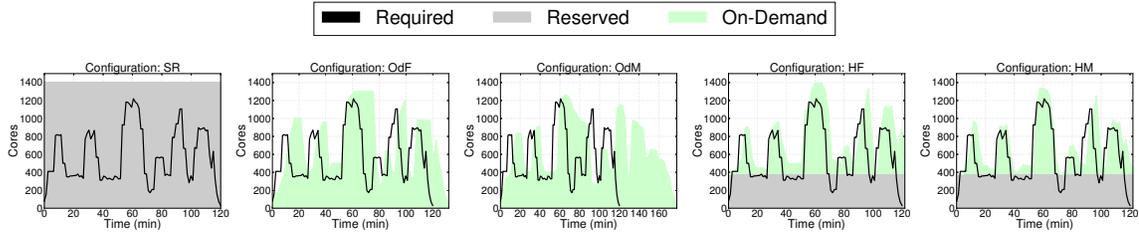


Figure 8.17: Resource allocation graphs for the five provisioning strategies in the case of the high variability scenario.

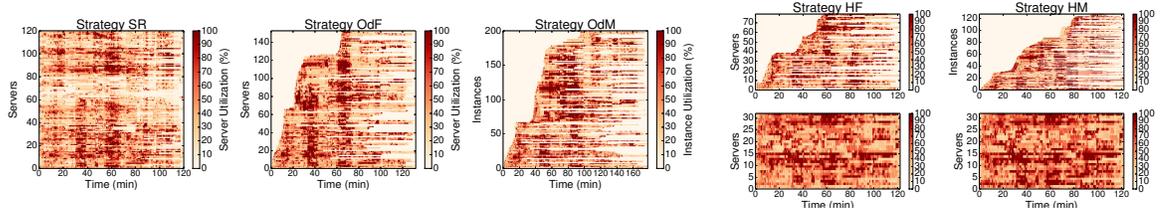


Figure 8.18: Resource utilization for the high variability scenario across the five provisioning strategies. For strategies HF and HM we separate reserved (bottom) from on-demand (top) resources.

of variability in the quality of an instance, and of the high instance churn (releasing instances immediately after use), due to their poor behavior. 43% of obtained instances were released immediately after use. The hybrid strategies (HF and HM) provision reserved resources for the minimum, steady-state load and use on-demand resources beyond that. Spin-up overheads induce some delay in the completion of the scenario, although this only amounts to 2.6% over SR. With HM, this delay is primarily due to instances that misbehaved and were immediately released after the completion of their jobs, requiring resources to be obtained anew (about 11% of instances). This issue is less pronounced when all on-demand resources are large instances (HF).

Figure 8.18 shows the CPU utilization of each instance throughout the execution of the high variability scenario for the five provisioning strategies. CPU utilization is sampled every 2 seconds and averaged across the cores of an instance. For the hybrid strategies we separate the reserved from the on-demand resources. For the on-demand resources, instances are ranked in the order in which they are obtained.

In the case of the fully reserved provisioning strategy (SR), a small fraction of the

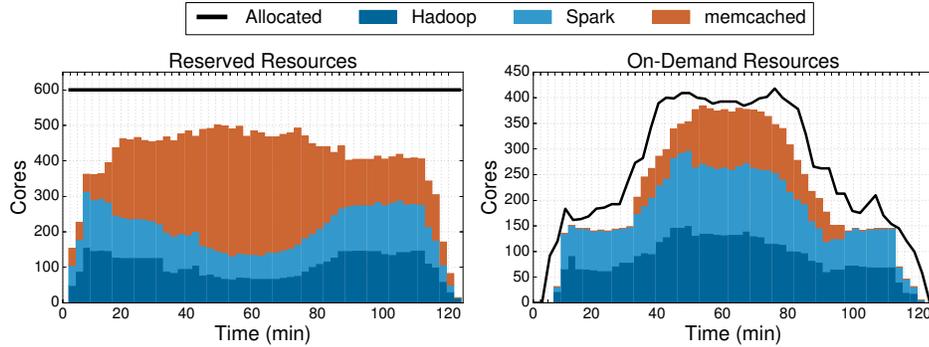


Figure 8.19: Breakdown of allocation per application type.

instances operate at high utilization as we try to co-schedule as many applications as possible, however, the majority of instances are greatly underutilized. This is consistent with the findings of Figure 8.17a; because resources are provisioned for peak load most of the machines operate at low utilization when load is lower than the maximum. The majority of resources are used only during the two load surges at 32 and 60 minutes. Strategies OdF and OdM obtain resources as they become necessary (shown by the fact that not all instances exist at time 0). Although the total number of instances used during the execution of the scenario with OdF is similar to SR, most instances are released when no longer needed, hence the number of active instances during off-peak load is significantly lower. In the case of the OdM strategy instances are additionally released when they behaved poorly for a given application. Note that because of the high instance churn, the total number of instances used throughout the execution of the scenario is higher for OdM than for OdF. The scenario also takes longer to complete for OdM (178 as opposed to 120 minutes).

Finally the hybrid strategies maintain the utilization of reserved resources high throughout the execution of the workload scenario, and obtain on-demand resources as needed. HF needed in total 72 on-demand instances, although only 34 of those are used on average. More than 60 on-demand instances are only used during load surges. HM needs a higher number of on-demand resources, because it also uses smaller instances, and because poorly-performing instances are released and replaced by new ones. Note that the fraction of released instances due to poor performance is lower for HM than for OdM, since only jobs that can tolerate some performance

unpredictability are mapped to smaller on-demand instances. Both hybrid strategies complete the scenario in the same time as SR.

Figure 8.19 breaks down the allocation of the low variability scenario by application type, for strategy HM. Initially the reserved resources are used for most applications, until load reaches the soft utilization limit. Beyond that, the interference-sensitive memcached occupies most of the reserved resources, while the batch workloads are mostly scheduled on the on-demand side. When the increase in the memcached load exceeds the capabilities of the reserved resources, part of the latency-critical service is scheduled on on-demand resources to avoid long queueing delays, although it is often allocated larger on-demand instances to meet its resource quality requirements.

8.5.5 Additional Provisioning Considerations

Spot instances: Spot instances consist of unallocated resources that cloud providers make available to users through a bidding interface. Spot instances do not have availability guarantees, and may be terminated at any point in time if the bidding price is lower than the market price for an instance type. Incorporating spot instances in provisioning strategies for non-critical tasks or jobs with very relaxed performance requirements can further improve the system’s cost-efficiency.

Reducing unpredictability: Resource partitioning (e.g., cache or network bandwidth partitioning) has the potential to improve isolation between instances sharing one or more resources, thus reducing performance unpredictability in fully on-demand provisioning strategies. We plan to investigate how resource partitioning complements provisioning decisions in future work.

Data management: In our current infrastructure both reserved and on-demand resources are in the same cluster. When reserved resources are deployed as a private facility, the provisioning strategy must also consider how to minimize and manage data transfers and replication across the private and on-demand resources.

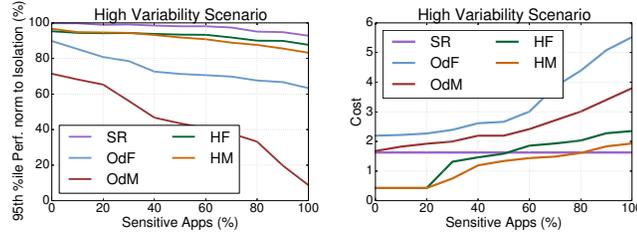


Figure 8.20: Sensitivity to application characteristics.

8.5.6 Sensitivity to Workload Characteristics

We now evaluate how the performance and cost results change as the characteristics of the applications change with respect to how sensitive they are to interference. Figure 8.20 shows the 95th percentile of performance for the five strategies as the percentage of jobs that are sensitive to interference increases. We modify the high variability scenario used before, such that the number of jobs that cannot tolerate performance unpredictability increases. In the left-most part of the graph, most jobs are batch Hadoop applications, which can tolerate some resource contention; as we move to the right part of the graph the majority of jobs are latency-critical memcached applications and real-time Spark jobs.

The statically-provisioned strategy (SR) behaves well even when most applications need resources of high quality, as it is provisioned for peak load, and there is no external load. The two hybrid strategies also behave well, until the fraction of sensitive applications increases beyond 80%, at which point queueing in the reserved resources becomes significant. The purely on-demand strategies are the ones that suffer the most from increasing the fraction of sensitive applications. OdF and especially OdM significantly degrade the performance of scheduled applications, both due to increased spin-up overheads, and because more applications are now affected by external contention.

With respect to cost, increasing the fraction of applications that are sensitive to interference impacts all strategies except for SR. Since HF and HM can use the reserved resources for the sensitive jobs, their cost increases only beyond the 30% mark, at which point more on-demand resources have to be purchased to avoid increased

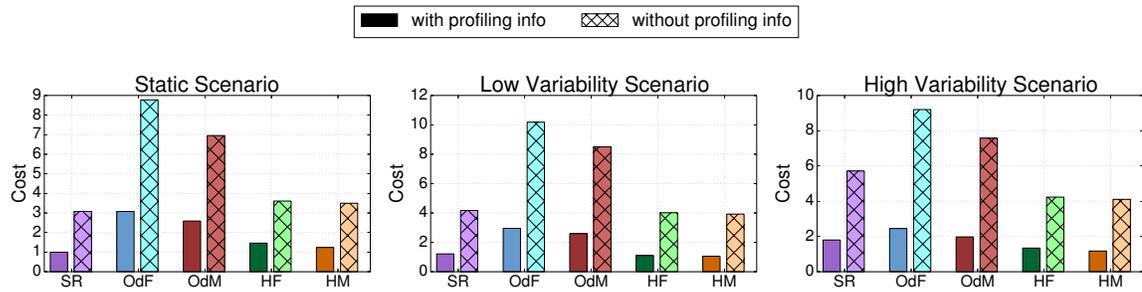


Figure 8.21: Cost of the three workload scenarios with and without the profiling information from Quasar.

queueing in the reserved resources. The two on-demand strategies experience a significant cost surge, as increasing the fraction of sensitive applications results in a lower degree of co-scheduling and the need for new resources.

8.5.7 Cost Impact of Information from Quasar

Finally, we examine how removing the information on the resource preferences of new jobs affects the cost of the five provisioning strategies. In this case, latency-critical applications, such as memcached are provisioned for their peak load, and batch jobs (Hadoop and Spark) use the default framework parameters, for example for the number of tasks per core, heapsize, etc. Figure 8.21 shows the cost with and without the information from Quasar for the three workload scenarios. Since overprovisioning is now much more prominent both the statically-reserved and the on-demand strategies incur significantly higher costs. The differences become more pronounced for scenarios with load variability, where overprovisioning is higher. For most cases the relative ordering between strategies remains the same; for example in the high variability scenario even without the information from Quasar the hybrid strategies have lower cost than SR and significantly lower than the on-demand strategies.

8.6 Related Work

Cluster management: The increase in the size and number of large-scale DCs has motivated several designs for cluster management. Systems like Mesos [139], Torque [259] and Omega [232] all address the problem of allocating resources and scheduling applications in large, shared clusters. Mesos is a two-level scheduler. It has a central coordinator that makes resource offers to application frameworks, and each framework has an individual scheduler that handles its assigned resources. Omega on the other hand, follows a shared-state approach, where multiple concurrent schedulers can view the whole cluster state, with conflicts being resolved through a transactional mechanism [232]. Dejavu identifies a few workload classes and reuses previous resource allocations for each class, to minimize reallocation overheads [266]. CloudScale [240], PRESS [118], AGILE [203] and the work by Gmach et al. [117] predict future resource needs online, often without a priori workload knowledge. Finally, auto-scaling systems, such as Rightscale [224], automatically scale the number of physical or virtual instances used by webserving workloads, to accommodate changes in user load.

A second line of work tries to identify the specific resources that are appropriate for incoming tasks [76, 188, 199, 285]. Paragon uses classification techniques to determine the impact of platform heterogeneity and workload interference on an unknown, incoming workload [76]. It then uses this information to schedule each workload in a way that enables high performance for the job and high utilization for the cluster. Paragon, assumes that the cluster manager has full control over all resources, which is often not the case in public clouds. Nathuji et al. developed a feedback-based scheme that tunes resource assignments to mitigate interference effects [200]. Yang et al. developed an online scheme that detects memory pressure and finds colocations that avoid interference on latency-sensitive workloads [285]. Similarly, DeepDive detects and manages interference between co-scheduled workloads in a VM environment [204]. Finally, CPI2 [296] throttles low-priority workloads that induce interference to important services. In terms of managing platform heterogeneity, Nathuji et al. [199] and Mars et al. [186] quantified its impact on conventional benchmarks and Google

services, and designed schemes to predict the most appropriate server type for each workload.

Hybrid clouds: Hybrid clouds consist of both privately-owned and publicly-rented machines and have gained increased attention over the past few years for several reasons, including cost-efficiency, as well as security and privacy concerns [20, 44, 143, 159, 294]. Breiter et al. [44] describe a framework that allows service integration in hybrid cloud environments, including actions such as overflowing in on-demand resources during periods of high load. Farahabady et al. [143] present a resource allocation strategy for hybrid clouds that attempts to predict the execution times of incoming jobs and based on these predictions generate Pareto-optimal resource allocations. Finally, Annapureddy et al. [20] and Zhang et al. [294] discuss the security challenges of hybrid environments, and propose ways to leverage the private portion of the infrastructure for privacy-critical computation. The provisioning strategies discussed here are also applicable to hybrid clouds.

Cloud economics: The resource pricing of cloud providers has been extensively analyzed. Ben-Yehuda et al. [36] contest whether the pricing strategy of spot instances on EC2 is indeed market-driven, and discuss alternative pricing strategies. Deelman et al. [71] discuss provisioning strategies for a single astronomy application on a cloud provider. Li et al. [172] compare the resource pricing of several cloud providers to assist users provision their applications. Finally, Guevara et al. [127] and Zahed et al. [290] incorporate the economics of heterogeneous resources in market-driven and game-theoretic strategies for resource allocation in shared environments.

8.7 Conclusions

We have discussed the different provisioning strategies available on cloud providers today and showed their advantages and pitfalls with respect to cost, performance predictability and initialization overheads. We have also designed two new hybrid provisioning strategies, that use both reserved and on-demand resources, and leverage the information on resource preferences of incoming jobs and the quality of previously-obtained on-demand instances, to map jobs to reserved versus on-demand resources.

We showed that hybrid provisioning strategies can provide the best of both worlds in terms of performance and cost-efficiency; they preserve QoS for the majority of jobs, improve performance by 2.1x compared to fully on-demand resources, and reduce cost by 46% compared to fully reserved resources.

Chapter 9

Conclusions and Future Work

This dissertation has presented scheduling and resource management techniques that enable resource-efficient and performance-aware datacenters. In particular we have made the following contributions.

- **Big Data in System Management:** We have designed two systems that leverage data mining techniques to quickly extract the resource preferences of previously-unknown applications. First, Paragon (Chapter 3) uses a recommender system based on collaborative filtering to determine the most suitable hardware platforms for a new workload, and the sensitivity it experiences to interference in shared resources (Chapter 5). Subsequently, Quasar (Chapter 4) generalizes this framework to tackle the more general problem of cluster management in datacenters, addressing both resource assignment (type of resources), and resource allocation (amount of resources). Leveraging data mining not only improves application performance and cluster-wise utilization by 2-3x, but enables practical management solutions at the scale of thousands of machines.
- **High-Level Declarative Interfaces:** We have highlighted the performance and efficiency pitfalls of reservation-based interfaces in datacenters. In Quasar (Chapter 4), we have proposed a high-level, declarative interface that centers around performance. Users specify the performance (QoS) target a new application must meet and the system translates it to resources using the data mining approach

detailed above. The declarative interface simplifies the responsibility of the user, while allowing flexibility to the cluster manager to better allocate resources.

- **Scalable Scheduling Techniques:** We have developed two systems that enable scalable scheduling in the presence of heterogeneous resources. First, we presented Tarcil (Chapter 7), a scheduler that bridges the gap between centralized systems that optimize decision quality, and distributed schedulers that optimize decision latency. Tarcil relies on a simple analytical framework, that provides statistical guarantees on the quality of allocated resources. Second, we designed HCloud (Chapter 8) which enables efficient resource provisioning in public clouds. Both systems enable millisecond-level scheduling decisions at high cluster load, while guaranteeing per-application performance constraints.

We believe that these contributions open several interesting directions for future work. Our work on applying data mining principles to datacenter management places an emphasis on the importance on finding practical solutions for large-scale system challenges. We have shown that contrary to the traditional trial-and-error approach, mining the knowledge systems accumulate through data collection in a mindful fashion can provide invaluable insight on application requirements, which translates to both performance and efficiency benefits. We hope that architects and system designers will apply this approach to other large-scale problems, including managing dependencies across multi-tier services, performance debugging for distributed applications, and design of heterogeneous and reconfigurable systems. Additionally, while we have so far focused on improving resource efficiency at the cluster management level, efficiency is sacrificed in lieu of performance across the system stack. Designing hardware and software schemes that guarantee strict isolation between applications sharing system resources can further improve resource efficiency. Finally, a lot of performance unpredictability comes from the many levels of indirection in the software stack. Providing feedback to application designers on application-level inefficiencies can bridge the programmability-performance gap and improve predictability in large-scale systems. We leave these endeavors to future work.

Appendix A

Storage Modeling of Datacenter Workloads

A.1 Introduction

With the advent of social networking and cloud data-stores, user data is increasingly being stored in large-capacity and high-performance storage systems. These systems account for a significant portion of the total cost of ownership (TCO) of a datacenter (DC) [229, 69]. Specifically, for online services, data retrieval is often the bottleneck to application performance [230, 229], making efficient storage provisioning a first-order design constraint. One of the main challenges when trying to evaluate storage system options is the difficulty in replaying the entire application in all possible system configurations. The effort itself can be highly inefficient from the time and cost perspective, given the scale of DC deployments (hundreds of TBs, over tens of thousands of servers). It is hence imperative to invest in frameworks that allow for extensive workload analysis, characterization and modeling.

Large-scale Online Services differ from conventional applications in that they cannot be approximated by single machine benchmarking, due to patterns that emerge from user behavior in a large-scale environment. Furthermore, privacy concerns make source code, user behavior patterns and datasets of DC applications rarely available to storage system designers. This makes the development of a representative model

that captures key aspects of the workload’s storage profile, even more appealing. Once such a model is available, creating a tool that reproduces the application’s storage behavior via a synthetic access pattern will enable large-scale storage studies, decoupled from the requirement to access application code.

Despite the merit in this effort, previous work on I/O workload generation lacks the ability to capture the spatial and temporal locality of I/O access patterns, causing them to significantly deviate from the application’s real characteristics. In this chapter, we provide a framework for research on large-scale storage systems that addresses these issues. This infrastructure includes probabilistic, state diagram-based models that capture information of *configurable granularity* on the workload’s access patterns. We develop the models from production traces of real DC applications based on previous work [230]. We extend these models to a granular, hierarchical representation in order to identify the optimal level of detail for each application. Then we perform an in-depth, per-thread characterization of the storage activity of ten large-scale DC applications in terms of the functionality, intensity and fluctuation of I/O behavior. To the best of our knowledge this is the first study at a per-thread granularity for large-scale applications, including information on their spatial locality. Furthermore, we design a tool that recognizes these models and recreates synthetic access patterns with I/O features that closely match those of the original applications. We perform extensive validation of our methodology to ensure resemblance between original and synthetic loads in both I/O characteristics and performance metrics.

The main features we introduce for accurate I/O generation are:

1. The ability to issue I/Os with specified inter-arrival times, both static and following time distributions.
2. The ability to preserve the spatial and temporal locality of I/O accesses, as well as the features and weights of each transition in the state diagram.
3. The ability to modify the intensity of the generated I/Os by scaling the inter-arrival time of I/O requests. This enables high performance storage systems evaluations (e.g., Solid State Drives).

We use our methodology (model and tool) to evaluate two important DC storage design challenges. Firstly, we explore the applicability of Solid State Devices (SSD) caching in DC workloads. Using our tool, we show that for most of the examined DC applications, SSD caching offers a significant storage system speedup without application change (31% on average for a 32GB SSD cache). In the second use case, we motivate the need for defragmentation in the DC. We observe that user data gets accumulated over a period of time and files get highly fragmented. Using this information from tracing [97], we rearrange blocks on disk in order to improve the sequential characteristics of the workloads. Using the tool to run the defragmented traces we show that defragmentation offers a significant boost in performance (18% on average), in some cases even greater than incorporating SSDs.

Succinctly, the main contributions of this work are:

- We present a concise statistical model that accurately captures the I/O access pattern of large-scale applications including their spatial locality, inter-arrival times and type of accesses. It is also hierarchical, which allows configurable level of detail to accommodate the features of each application.
- We implement a tool that recognizes this model and recreates synthetic access patterns with same I/O characteristics and performance metrics as in the original application. No previous storage tool (e.g., IOMeter) can simulate spatial and temporal locality of DC workloads.
- This methodology enables storage system studies that were previously **impossible without full application deployment and without access to the real applications**. We demonstrate the applicability of our tool in evaluating SSD caching and defragmentation. These spatial locality-based studies have been unexplored due to lack of a tool that allowed their evaluation.

A.2 Related Work

Significant prior work [163] has studied how to efficiently provision DC storage systems. However, a necessary requirement towards efficiently configuring the storage

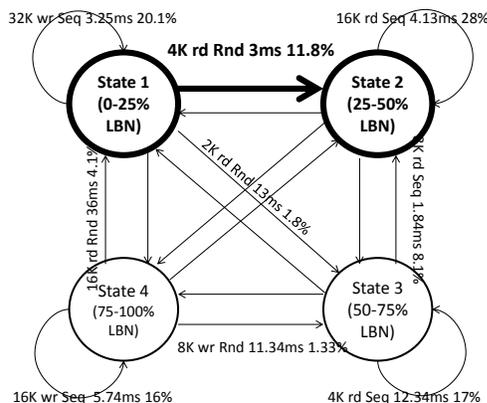


Figure A.1: One level state diagram

system is studying DC workloads. A convenient approach for that should involve a model that captures the workload’s representative features and a tool that accurately recreates its access patterns.

Despite this, prior large-scale storage configuration techniques are primarily empirical, based on the workload’s characteristics as derived from traces [156]. Kavalanekar et al. [156, 155] use a trace-based approach to characterize large online services for storage system configuration and performance modeling. Traces offer useful insight on the characteristics of large-scale workloads, but their usefulness is limited by the system upon which they have been collected. Regenerating I/O workloads with high fidelity can offer far richer information towards understanding the behavior of workloads whose implementation remains largely unknown. It also enables addressing instrumental challenges in storage system design (e.g., incorporating SSDs, defragmentation, placement/migration of hot data) when optimizing for performance and efficiency.

IOMeter [147], SQLIO [245], Vdbench [264] are all open-source generators of disk I/O loads. IOMeter allows for specific I/O characteristics to be defined, SQLIO simulates aspects of the disk load of the Microsoft SQL Server, while Vdbench apart from the feature of I/O generation is equipped with the capability of trace replay. Finally, a workload generator relying on online histograms in a virtual machine over VMWare’s ESX Server [8] captures information on disk I/O without significant CPU or latency overheads. However, all these workload generators lack the ability to

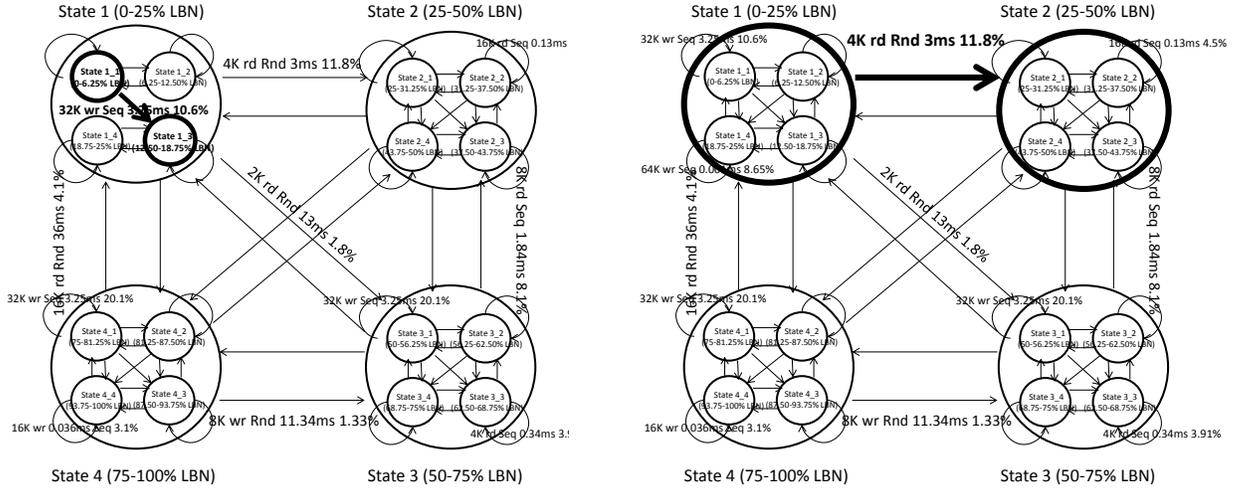


Figure A.2: Two level state diagram: (a) Transition between minor states, and (b) Transition between major states

exploit the temporal and especially the spatial locality of DC applications. Spatial and temporal locality are extremely important for DC applications, due to the different behavior of storage devices with and without locality. Ignoring locality can result in greatly misleading results in performance, power and TCO.

Finally, where applicable, these tools are based on outstanding I/Os instead of inter-arrival times. However, the latter offers a better representation of the workload’s behavior [197], decoupled from the system that hosts it. For our work, we extend the functionality of DiskSpd [90], an I/O workload generator, in ways that enable us to recreate representative DC loads.

A.3 Modeling and Generation Process

A.3.1 Basic State Diagram Model

Our approach requires a model that captures the I/O features and locality of storage activity from the application’s point of view. This means that the model needs to cluster accesses based on their spatial locality and characteristics. For this purpose we use the Markov Chain representation proposed by Sankar et al. [230]. The models are

trained based on real storage traces from production servers of a large-scale datacenter deployment, and can capture I/O accesses in one or multiple servers, and from one or multiple users. According to the model, states correspond to ranges of logical blocks on disk (LBNs) and transitions represent the probabilities of switching between LBN ranges. Each transition is characterized by a set of features that reflect the workload’s I/O behavior and consist of the block size, randomness, type of I/O (read, write) and inter-arrival time between subsequent requests.

The insight behind the model’s structure is that spatial locality is represented by the clustering of I/Os corresponding to the same state, and temporal locality (i.e., subsequent I/Os) is represented by the transitions between states in the model. Therefore, it provides a comprehensive and modular representation of the workload’s I/O behavior. The probability for each transition is calculated as the percentage of I/Os that correspond to it. Figure A.1 demonstrates a simplified form of the state diagram with four states, each of which corresponds to 25% of the total LBNs. The model works as follows (highlighted part of the diagram): If an I/O corresponds to State 1, there is an 11.8% probability that the next I/O will be a 4K read, random access with an inter-arrival time of 3ms that corresponds to State 2.

A.3.2 Hierarchical State Diagram Model

Different applications have different access patterns, some requiring more detail than others to be accurately captured. To convey information of finer granularity, we have extended the previous model to a hierarchical representation. Figure A.2 demonstrates one such model with two levels. To build a two-level model each state in the one level diagram is subdivided in four states and becomes a new state diagram. The two-level diagram will have 16 states. Here LBNs are divided in four states. In general, the number of states per level is chosen such that the probabilities of transitions are minimized.

Perhaps counter-intuitively, the number of transitions in the new diagram is not 256 but 76. As shown in Figure A.2, level-two (fine-grained) transitions only exist within the large states but not across them. This means that a transition happens

Levels	State Count	Transition Count	
		Hierarchical Model	Flat Model
1	4	16	16
2	16	76	256
3	64	316	4096
4	256	1276	65536
5	1024	5116	1048576
10	1048576	5242876	109951162776

Table A.1: Scalability of the model in terms of number of states and transitions with an increasing number of levels.

either between two minor states (Figure A.2(a)) or between two major states (Figure A.2(b)). This approach exploits the fact that spatial locality is mostly confined within states. The number of transitions for a given level is given by

$$4^{l-1}16 + \sum_{i=1}^{l-1} 4^{i-1}12 \quad (1)$$

while for the flat model it is given by 16^l , where l is the number of levels.

Table A.1 shows how the number of states and transitions scales for up to 10 levels. It becomes obvious that for the flat representation the number of transitions increases exponentially with the number of states, while the hierarchical model has a linear relation with the state count. This choice does not cancel the value of a flat model, but rather proposes that a hierarchical model is just as beneficial without making the number of transitions intractable. Comparing the throughput of models constructed with the hierarchical and the flat representations shows less than 5% difference in throughput.

The number of levels reflects the complexity of an application’s access pattern and as shown in the validation section (Section A.4), finer granularity is instrumental to accurately represent some applications. The proposed model structure guarantees scalability even for applications that require many levels.

A.3.3 Storage Activity Fluctuation

It is well known that DC applications experience high fluctuations in their activity, with peak activity usually occurring throughout periods of the day and low activity be

present throughout the night. In order to generate representative storage workloads of real applications one must account for this effect. Studying the fluctuation of the storage activity for DC applications reveals that the main feature that changes is the intensity of the I/O requests, i.e., throughout specific periods of an application's lifecycle inter-arrival times decrease significantly. Other features however, like the spatial locality and size of the requests are self-similar throughout this lifecycle [197]. In order to account for this fluctuation, we calculate the inter-arrival times over shorter periods of activity and progressively switch between workload intensities. Essentially each model is composed by multiple models of different intensities that capture the transient features of the I/O requests.

A.3.4 Generation Tool Design - DiskSpd

The model, previously discussed, is the first step in recreating accurate DC I/O loads. The second step, involves a tool that recognizes the model and generates storage workloads with high fidelity, using some configuration knobs.

For this purpose we use DiskSpd, a tool that started as a means to measure disk I/O bandwidth and expanded to a complete workload generator [90]. It performs read and/or write I/Os in burst mode on either disks or files, given the I/Os' block size, randomness, and initial block offset. The former consist of a subset of the most relevant features of DiskSpd for the current study. Other features include controlling system parameters such as hardware or software (OS) caches, thread affinity, number of outstanding I/Os, etc.

To recreate a representative workload using the model previously discussed, we have introduced a series of features in DiskSpd. The following subsections describe these features.

Inter-arrival Times (Average and Distributions)

Studying real application traces has shown that burst mode I/O accesses, though present for short periods of time, are not the norm and do not dominate an application's lifetime. Subsequent I/Os tend to have well-defined time margins between

them. Narayanan et al. [197] have shown that inter-arrival times are a critical feature of I/O behavior, especially in DC applications that experience high peaks and low troughs throughout their execution. Multiple studies quantify the magnitude of this metric and explore the differences among DC workloads [230, 229].

To demonstrate these margins between accesses of specific block ranges, we implement inter-arrival times in DiskSpd, which are calculated for each transition and measured in ms. Enabling inter-arrival times also means disabling the simultaneous tuning of outstanding I/Os since the two are incompatible, with the former ensuring an "idle" period of time between I/Os and the latter ensuring a defined number of on-the-fly I/Os in the queues.

The use of inter-arrival times instead of outstanding I/Os in a workload generator is *first proposed here*. Previous tools are based on defining the system's queue length. However, queued I/Os do not characterize an application as well as inter-arrival times, since queue length is a system feature, while I/O intensity a workload feature. The difference between the two becomes clearer in the case where we want to create a more intense workload as described in Section A.3.4.

Furthermore, in order to capture the variations in storage intensity we have added the feature of inter-arrival time distributions, i.e., during the workload's execution inter-arrival times can follow one of the following distributions: normal, exponential, Poisson and gamma. This permits a closer resemblance to the fluctuations of a workload's intensity throughout its lifetime, as well as the capture of burst I/Os.

In the default version of the tool, the mean for normal distribution (μ), the rate parameter for exponential (λ), the expected number of occurrences for Poisson (λ) and the scale parameter (θ) for gamma correspond to the mean inter-arrival time calculated from the traces.

Multiple Threads and Thread Weights

Access patterns of real applications have distinct per transition characteristics. In order to recreate an I/O load using the state diagram model, we have added the feature of executing threads with different I/O characteristics each (block size, randomness, type (rd/wr), target LBN range and inter-arrival times). Each thread corresponds to

a transition in the state diagram and maintains the original access pattern via the notion of *thread weight*, i.e., the proportion of I/O accesses that correspond to each thread. During the threads' execution, we ensure that thread weights are satisfied with less than 0.05% deviation from the target weights by adjusting their "idle" time.

This mechanism might seem redundant if one considers thread weights as a straightforward translation of inter-arrival times. Although inter-arrival times are a strong indication of the proportions of accesses for a transition, there are cases where fast I/Os are not common, but are confined in a short period of the application's execution. In this case, simply maintaining inter-arrival time will not ensure the transition's weight. Although these events are rare, this mechanism ensures that thread weights are maintained in all cases.

Furthermore, in order to guarantee that thread weights are satisfied throughout the workload's execution we perform a Round Robin visit in states so that all threads are active in different phases during the program's execution instead of limiting them in an arbitrary period of activity. This way the synthetic trace becomes a compressed version of the original workload. Although this does not cover all possible transition patterns, self-similarity tests [156] in original DC applications have verified that indeed the spatial characteristics of I/Os are consistent across time.

Intensity Knob

One of the main objectives behind developing this tool is to evaluate different storage system configurations. Although when referring to disks, inter-arrival times are within a few milliseconds or tenths of a millisecond, when switching to SSDs that number is expected to fall dramatically, since I/Os are expected to arrive at a higher rate. Current production traces do not have such intensity; however, we expect workloads to be tuned to faster storage systems using SSDs. In order to replay workloads compatible with such systems, we have added an intensity knob that scales inter-arrival times down or up to increase or decrease their intensity respectively. This feature clarifies the distinction between outstanding I/Os and inter-arrival times. Queuing more I/Os does not emulate a faster storage system, since in an SSD-based system, for example, I/Os do not simply get queued in larger numbers, they also get

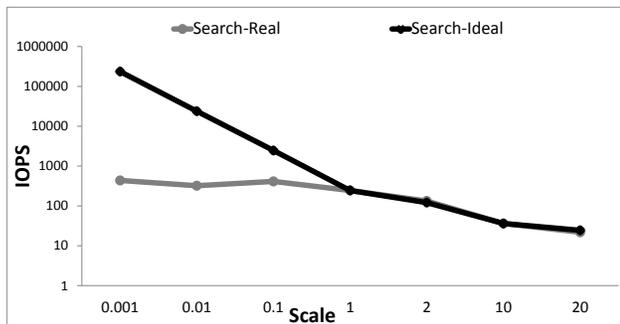


Figure A.3: Throughput scaling for different workload intensities.

serviced faster. Maintaining the outstanding I/Os queue length in this case, stresses the storage system out of proportion and is not useful for DC scaling studies. Having this knob offers the opportunity to evaluate a storage system configuration based on the workload’s expected intensity margins. It also enables studies that scale the number of users that initiate requests in the system.

For example, in a hard disk-based system (HDD) when intensity exceeds the system’s queues’ capabilities, throughput levels-off. Figure A.3 shows this inability of the HDD system to service high-rate requests. Smaller scale corresponds to a more intense workload. Although we assume that I/Os will not be dropped, unless timeouts are present in the system, the application can still not meet its increased performance requirements. The use of SSDs for storage is motivated, among others, by this performance limitation of the HDD-based system.

An important note to make here is that our work is based on an open-loop approach, which means that applications are not retuned when we switch to an SSD-based system. This potentially underestimates the benefit from the use of SSDs, but offers a more concise comparison between the capabilities of the storage systems, since all other parameters remain constant. A second assumption is that, in order to use the same models as in the HDD-based system, we expect subsequent I/Os to be independent of each other, therefore scaling the inter-arrival times is a valid approximation of the workload’s access pattern when run on a faster storage system. This assumption is justifiable for large-scale applications where most requests come from

Thread Type	Functionality	Intensity	Fluctuation
Data #0	Data	High	High
Data #1	Data	High	Low
Data #2	Data	Low	High
Data #3	Data	Low	Low
Log #4	Log	High	Low
Log #5	Log	Low	High

Table A.2: Per-thread classification for the ten examined applications based on *query type*, *intensity* of storage activity and *fluctuation* in the intensity of I/O requests.

different users. Therefore, the intensity knob makes the application more compatible with a high service rate storage system, while retaining the previous spatial locality. Whether this locality and hence the model is subject to change in a faster system is deferred to future work.

A.4 Characterization and Validation

A.4.1 Original DC Workloads

For all our experiments we use traces from production servers of ten popular large-scale DC applications. Messenger, Display Ads and User Content are the SQL portions of an Online Messenger, an Ads Display and a Live Storage application respectively. For each one, we study the part that maintains the SQL database with user information. These applications service thousands of users; therefore the data being accessed is typically spread across most of the provisioned disks.

Email, Search and Exchange are latency-critical online services. Email and Exchange are hosted in a much larger storage system than Search. Search has significant spatial locality, with some portions of the disk being frequently accessed and others heavily underutilized. TPCC, TPCE and TPCB are large-scale databases, part of the TPC benchmark suite [260]. Finally, D-Process is a highly-parallelized distributed computing application that resembles MapReduce [69], collecting and processing large amounts of information on applications such as Search. Its storage comprises of a large number of disks, partitioned between data and logs. These applications cover

the majority of large-scale workloads in modern DCs.

A.4.2 Generating Models from Traces

The first step in order to create the workload models is collecting real, 24-hour long (unless otherwise specified) traces from production servers, hosting the applications previously discussed. The I/O traces are collected from individual servers, however due to load balancing in DCs the storage behavior is similar across multiple machines [197]. The length of the traces is sufficiently representative of an application’s behavior, given the self-similarity of access patterns in DC workloads [197].

The traces are collected using ETW [97], which aside from information on I/O features (block size, type of request, etc.), tracks the file name, thread id and values of timestamps for each storage access. Having these traces, we create state-diagram models with different number of levels. The models are created by clustering I/Os in states based on their spatial locality and then categorizing them based on size, type (read/write) and randomness (random/sequential). Each distinct I/O category becomes a different transition and based on the number of I/Os that belong to each transition we calculate its inter-arrival time and probability. These models are then used to create the synthetic workloads.

A.4.3 DC Application Characterization

We previously described a way to model the I/O characteristics of large-scale DC applications. Here we perform an in-depth per-thread characterization of the ten applications based on this model and provide insights on their behavior. To the best of our knowledge no such per-thread, storage activity categorization and characterization has been previously performed for DC applications. We separate the storage traces per-thread and define different thread types for each application based on activity fluctuation, intensity (I/O rate) and functionality of the thread (Data or Log - where applicable). The different thread types are shown in Table A.2.

In Table A.3 we show the per-thread storage characterization for one online service (Exchange), one SQL-based application (Messenger), and one of the TPC benchmarks

App	Thread Type	Load	R:W	%Seq I/Os	Spatial Locality				Avg IOPS	Latency (ms)		
					St1	St2	St3	St4				
Exchange (6h)	Total		R	1:2.6	13.7	42.3	55.8	1.9	0.1	134.0	3.7	
			W	1	2.3	35.2	60.4	4.3	0.0	36.8	5.9	
	Data#0		R	1:2	2.1	42.1	57.4	0.5	0.0	51.2	3.9	
			W	1	1.3	38.1	59.8	3.1	0.0	18.4	5.2	
	Data#1		R	3.9:1	11.8	47.6	50.1	2.3	0.0	5.5	4.1	
			W	2	2.8	48.9	50.1	1.0	0.0	34.1	0.5	
	Data#2		R	1:100	12.7	72.4	27.4	0.0	0.2	0.3	0.5	
			W	0	-	-	-	-	-	0	-	
	Data#3		R	2.4:1	13.3	21.0	68.8	10.2	0.0	0.02	3.9	
			W	100	12.7	72.4	27.4	0.0	0.2	0.3	0.5	
	Messenger (6h)	Total		R	2.8:1	9.3	31.5	22.8	45.5	0.2	255.1	8.09
				W	2.8	7.3	31.0	38.6	30.8	0.2	194.0	10.0
Data#0			R	1.8:1	11.4	31.6	32.4	36.0	0.0	13.8	9.4	
			W	1	11.4	32.6	10.8	56.7	0.2	69.3	3.4	
Data#1			R	3.3:1	5.3	65.8	21.9	12.3	0.0	5.5	9.8	
			W	3.3	4.2	71.8	19.6	10.6	0.0	4.2	10.8	
Data#2			R	1:2.1	13.6	43.5	36.6	19.0	1.9	1.0	8.4	
			W	1	8.4	41.2	23.8	35.6	0.5	0.32	10.0	
Data#3			R	10:1	10.8	43.5	38.9	17.6	0.0	1.0	8.7	
			W	2.1	13.6	46.3	41.8	8.8	3.1	0.67	4.8	
Log#4			R	1:99	12.1	9.0	61.9	22.1	6.0	0.08	1.3	
			W	1	8.5	79.0	21.0	0.0	0.0	8E-5	10.8	
Log#5		R	1:100	12.9	30.8	60.8	8.4	0.0	0.01	0.5		
		W	99	13.2	7.5	63.7	22.8	6.0	0.08	0.5		
TPCE (11min)	Total		R	10:1	8.1	89.8	8.4	1.7	0.3	24,694	5.55	
			W	10	6.0	92.5	5.4	2.1	0	22,546	6.0	
	Data#0		R	12.5	16.3	91.0	8.0	0.2	0.8	1,172	5.7	
			W	1	23.7	56.0	26.9	15	2.1	2,148	0.7	
	Data#1		R	7:1	20.6	92.4	7.4	0.2	0	1,897	4.6	
			W	7	15.3	96.0	4.0	0	0	1,659	5.3	
	Data#3		R	70:1	15.5	91.2	6.2	2.4	0.2	859.3	5.4	
			W	1	37.9	87.7	10	2.3	0	238	0.5	
	Log#4		R	1:99	11.7	88.8	10	1.0	0.3	12.4	0.5	
			W	70.4	15.6	90.6	7.2	2.4	0.1	847.3	5.4	
	Log#5		R	1:100	1.4	89.0	11.0	0	0	0.5	0.5	
			W	1	7.1	99	0	0	1	12.0	0.7	

Table A.3: Per-thread characterization for one Online Service (Exchange), one SQL-based application (Messenger), and one of the TPC benchmarks (TPCE). From left to right we show the fluctuation of storage activity, the I/O features in terms of read:write ratio, percentage of sequential I/Os, average block size and inter-arrival time and spatial locality, as well as average performance metrics, in terms of throughput and latency for each thread type.

(TPCE). We show the fluctuation for the aggregate workload and each thread type over the entire tracing period, as well as the I/O features in terms of average block size, inter-arrival time, read:write ratio, sequential I/Os and introduce here the study of *spatial locality*.

Spatial locality is estimated using one-level models, where each state corresponds to 25% of the machine's storage. The third to last column denotes the thread weight of the *individual thread* and of the *entire thread type*. We also show average performance metrics (throughput and latency). Examining *Exchange* reveals that the majority of storage activity comes from few, very I/O intensive threads (*Data #0*), while threads with no fluctuation, or low activity account for a considerably lower portion of the total throughput. *Exchange* is random, write I/O-dominated, while studying its spatial locality reveals that most accesses happen in the first half of the provisioned storage.

Email and Search and Exchange have similar behavior in terms of thread type classification and per-thread storage activity. Similarly, all the SQL applications (Messenger, Display Ads, User Content and D-Process) experience high resemblance in their storage activity. Finally, the I/O behavior of TPCC is very close to that of TPCE, with the DSS TPCH slightly deviating in terms of number of threads and intensity of storage activity.

A.4.4 Validation

Validating the accuracy of the model and the tool is necessary in order to ensure that original and synthetic workloads are similar in their storage activity. Apart from that, we want to identify the optimal level of detail for each application.

We perform the following steps:

1. Collect traces from production servers
2. Create workload models with a configurable number of levels
3. Run the synthetic workload and collect the new trace

Metrics		Original Workload	Synthetic Workload	Deviation
Messenger	Read:Write Ratio	2.8:1	2.8:1	0%
	% of Random I/Os	90.7%	89.4%	-1.38%
	Block Sizes	8K(87%) 64K(7.4%) 1K(1.6%)	8K(88%) 64K(7.8%) 1K(1.7%)	0.1-1%
	Thread Weights	T1 (19%) T2 (11.6%) T3 (1.6%)	T1 (19%) T2 (11.7%) T3 (1.6%)	0 - 0.05%
	Avg Inter-Arrival Time	4.63ms	4.78ms	1.1%
	Throughput (IOPS)	255.14	263.27	3.1%
	Average Latency	8.09ms	8.48ms	4.8%

Table A.4: Validation of I/O features & performance metrics for Messenger.

4. Compare I/O characteristics and performance metrics between the original and synthetic storage workload.

For the validation experiments we use a server provisioned for SQL applications, like Messenger and User Content, with 8 cores, 5 physical volumes, 10 disk partitions and a total of 2.3TB of purely HDD storage.

We maintain the configuration of the storage system the synthetic trace is replayed on, as close as possible to that of the original system by performing specific I/O requests in the appropriate disk partitions. For the SQL-based workloads, for example, Log I/Os are replayed in the Log partition while SQL queries are replayed in the data partition. For the remaining applications, the system varies in the real DC (Search and D-Process run on striped four-disk SATA systems); however, throughput does not greatly deviate from its expected value. Although this result might seem unexpected, for an incorrectly provisioned system, these applications have relatively low I/O throughput (IOPS) that is easily satisfied through a system engineered for SQL. Although the percentile difference is higher for these two applications, the absolute number of IOPS remains reasonably low.

For each one of the ten applications we evaluate the similarities in the features of the I/O requests (block size, rd/wr, rnd/seq, inter-arrival time and thread weight) as well as the performance metrics (throughput and latency) of the synthetic applications as opposed to the original ones. As far as the proportion of accesses is concerned, we verify that thread weights are satisfied with **less than 0.05%** deviation from their original values.

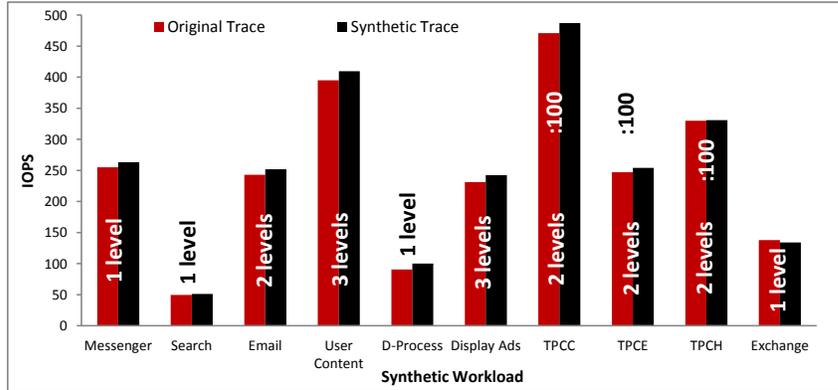


Figure A.4: Throughput comparison between original and synthetic workloads for the 10 applications.

Table A.4 shows the comparison for these metrics between original and synthetic workload for Messenger. The results are similar for the remaining applications. In all cases the deviation for the I/O features between original and synthetic load is less than 5%. Similarly for the performance metrics the deviation is at most 6.7% and on average 3.38%. Figure A.4 shows the throughput comparison between original and synthetic load for all applications. The difference in IOPS is always **less than 5%**, verifying the accuracy of the modeling and generation process. Furthermore, in order to ensure the consistency of our results, we calculate the variance between different runs of the same synthetic workload and guarantee a difference in throughput **less than 1%** in all cases.

Figure A.4 also plots the optimal number of levels per application, which is the one for which the synthetic trace resembles the original workload best. As optimal granularity we define the first number of levels for which the performance metrics stabilize (less than 2% difference in IOPS). That way, we convey the best possible accuracy with the least necessary model complexity. This methodology allows for a configurable level of detail in the model of each application. Figure A.5 shows how the throughput changes for each application for an increasing number of levels. In most cases one to three levels are sufficient for the I/O characteristics to stabilize.

Finally, we verify the resemblance in the fluctuation of storage activity between original and synthetic workloads. We perform a sensitivity study on the granularity at

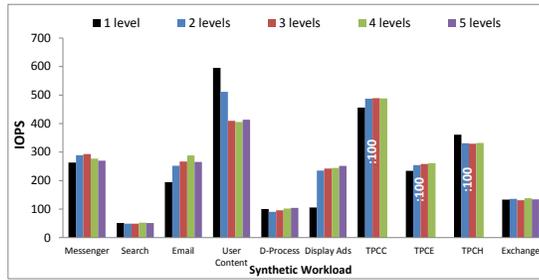


Figure A.5: Throughput for increasing number of levels.

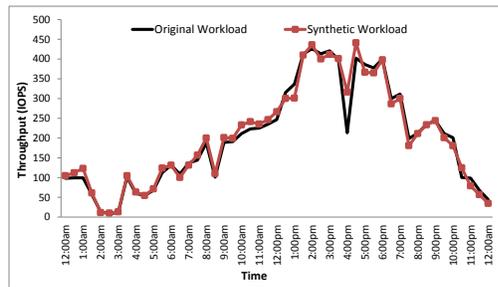


Figure A.6: Validation of storage activity fluctuation over 24h (Messenger).

which the I/O request intensity changes, to choose the interval over which we calculate inter-arrival times. We observe that within a 30-minute period there is no significant fluctuation in the storage activity for the examined applications. Figure A.6 shows the resemblance in storage activity fluctuation between the original and synthetic application for Messenger. The results are similar for the other applications as well. In all cases, peaks and troughs coincide for the two workloads, verifying that the activity fluctuation requirements are met.

A.4.5 Comparison with IOMeter

IOMeter is the most well-known open-source workload generator [147]. Although it offers many capabilities as far as access characteristics are concerned, it has limited information on the spatial locality of I/Os, making it unsuitable for several DC storage studies. Furthermore, IOMeter implements outstanding I/Os but cannot represent inter-arrival times, which seriously limits its intensity scaling capabilities. Finally, it does not allow specific file accesses, which as will be seen in Section A.5.2, would make it impractical to evaluate the benefits of defragmentation. Table A.5 summarizes the

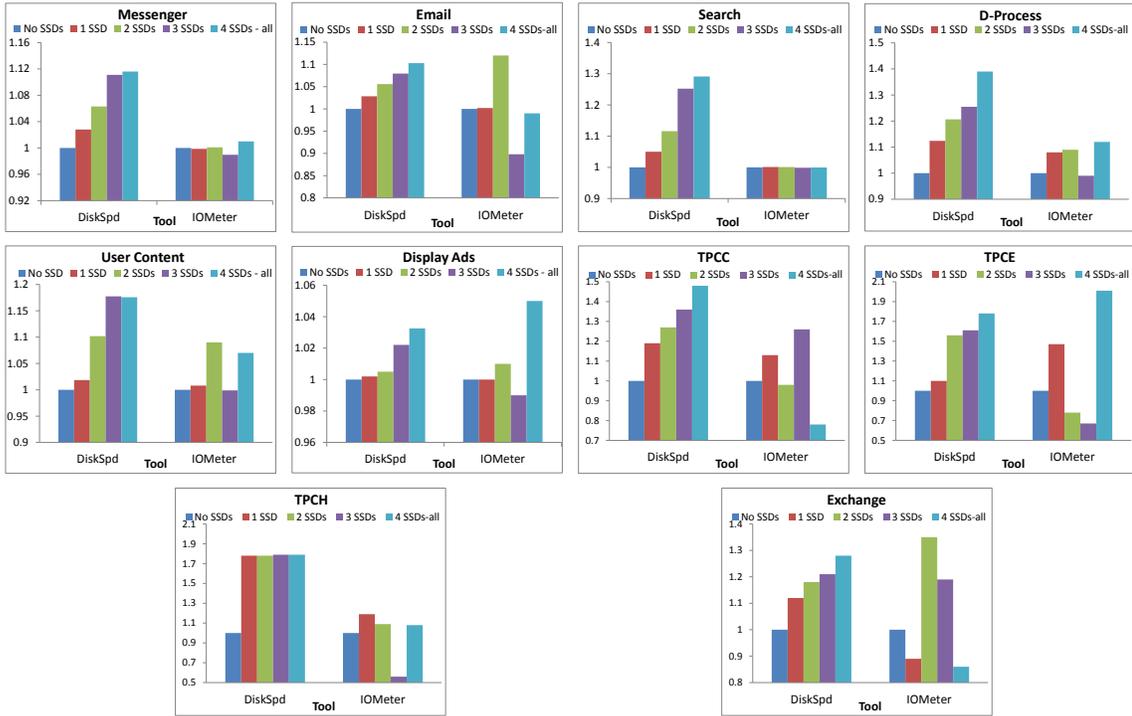


Figure A.7: IOMeter vs DiskSpd speedup comparison for (a) Messenger, (b) Email, (c) Search, (d) D-Process, (e) User Content (f) Display Ads (g) TPCC, (h) TPCE, (i) TPCH and (j) Exchange. The results for DiskSpd confirm the expected impact of SSD caching on workload performance. On the other hand, when the workloads are run using IOMeter there is either no performance speedup (e.g., Messenger) or inconsistent speedup (e.g., User Content) with an increasing number of SSD caches.

differences between the features supported by the two tools.

In this section we compare the performance characteristics of IOMeter and DiskSpd. For the purpose of this comparison no change is conducted in IOMeter, and the parameters for the tests are defined using the tool’s default knobs. We perform identical tests using both tools and quantify the difference in throughput and latency. The table below (Table A.6) shows how the tools behave in a series of simple access patterns with the exact same parameters. All tests are run for 30 seconds, performing I/O requests to a simple file. In the interest of clarity, we do not demonstrate all possible parameter configurations, but some representative examples. Note that no notion of spatial locality is introduced in these simple tests. From the results we observe that both tools behave similarly with a maximum throughput deviation of 3.4%.

Features	IOMeter	DiskSpd
Inter-arrival times (static or distributions)	No	Yes
Intensity knob	No	Yes
Spatial locality	No	Yes
Temporal locality	No	Yes
Trace replay	No	Yes
Granular I/O load detail	No	Yes
Individual file accesses	No	Yes

Table A.5: IOMeter vs DiskSpd features comparison

Test Configuration	IOMeter (IOPS)	DiskSpd (IOPS)
Block size: 4K, Inter-arrival time: 10ms, Read, Sequential	97.99	101.33
Block size: 16K, Inter-arrival time: 1ms, Read, Sequential	949.34	933.69
Block size: 64K, Inter-arrival time: 10ms, Write, Sequential	96.59	95.41
Block size: 64K, Inter-arrival time: 10ms, Read, Random	86.99	84.32

Table A.6: IOMeter vs DiskSpd comparison

The main difference in the two tools becomes evident when introducing the notion of spatial locality. To demonstrate that DiskSpd takes locality into account while IOMeter does not, we use an optimization technique that will be presented in more detail in the following section (Section A.5.1). SSD caching takes advantage of frequently-accessed blocks, and thus improves performance by avoiding accessing the disk often. If a tool takes into consideration spatial and temporal locality we expect an improvement in performance when the synthetic trace is run using SSD caches. We run the synthetic traces for the ten applications and the corresponding I/O tests that best resemble their behavior using IOMeter. No notion of spatial locality is incorporated in the latter. Figure A.7 shows how performance changes as we progressively add SSDs to the system for each of the workloads. The important point in these figures is not the precise speedup but the significantly different behavior of the tools. In all cases it becomes evident that IOMeter does not reflect the spatial and temporal locality of the original access pattern. For most applications there is no speedup for an increasing number of SSDs, due to incorrect caching of blocks, and for those that a speedup exists it is inconsistent with what would have been expected as caching becomes more intense (more SSDs - better speedup).

Workload	Spatial Locality - Level 1			
	<i>State1</i>	<i>State2</i>	<i>State3</i>	<i>State4</i>
TPCH	92.7%	6.2%	1.3%	0.0%
TPCE	89.8%	8.4%	1.7%	0.3%
D-Process	73.3%	18.8%	0.0%	0.0%

Table A.7: Spatial locality (SSD-caching study). Studying the spatial locality for the three applications with the highest benefit from SSD caching reveals that they have the highest I/O aggregation. This justifies the significant speedups from introducing SSD caching in the system.

A.5 Use Cases

One of the main benefits from using a modeling and generation tool for DC workloads is enabling storage studies, which would otherwise require access to application code or full application deployment. In this section we evaluate two possible use cases for the tool: SSD caching and defragmentation. Both are spatial and temporal locality-dependent, and have been unexplored using workload generation tools.

A.5.1 SSD caching

Designing an efficient storage system configuration for widely-deployed applications is a great challenge and in terms of proper provisioning, a field that separates high-quality systems from the norm. Studying the spatial locality of the ten DC applications reveals that for most of them, I/Os are aggregated in a small LBN range. This motivates incorporating SSDs to improve performance. Estimating performance, however, is not easy since writes in SSDs are highly unpredictable, and often as slow as disk, which makes performance gains greatly dependent on the I/O access features. This motivates using modeling and characterization to evaluate the potential benefit.

Due to the fact that we use an open-loop approach, applications are not retuned when switching to the SSD-based system. The experiments are performed by running the previous models on an SQL-provisioned server with 4 SSD caches (8GB each) [6], which we progressively turn on.

Figure A.7 shows the storage speedup when going from no SSD caches on (left bar) to all 4 SSD caches on (right bar). We observe that especially for the I/O



Figure A.8: File mapping (a) before, (b) during, and (c) after defragmentation. Blocks of different colors correspond to different files.

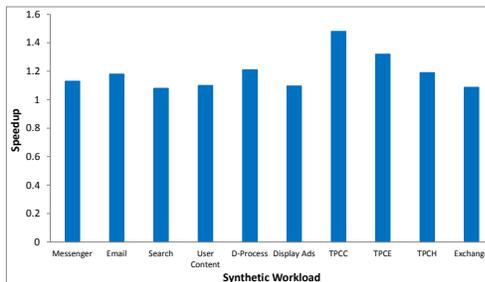


Figure A.9: Storage speedup from defragmentation

intensive TPCB, TPCE and D-Process, the performance benefit from using a large number of SSDs is significant (31% on average across all workloads for 4 SSDs and 79% maximum for TPCB). Studying the clustering of accesses in the corresponding models reveals that these three applications have the highest aggregation of I/O requests (Table A.7). Increasing the number of levels in this case, confines the accessed LBNs in a smaller range, thus better caching frequently-accessed blocks. An important note is that in Figure A.7, we refer to storage speedup and not speedup for the entire application. These workloads are not necessarily limited by storage but their performance improvement, and the expected improvement in efficiency, are strong incentives towards the use of SSD caching nonetheless.

A.5.2 Defragmentation

Most DC applications experience high levels of fragmentation, as user-requests get accumulated over time, with Random I/Os often exceeding 80%. This motivates the use of defragmentation to improve performance and efficiency. From the information provided by ETW [97] we can extract the name of the file for each I/O access, estimate fragmentation levels, and perform a block rearrangement to improve the sequential characteristics as shown in Figure A.8.

Workload	Read	Write	Before		After	
			Rnd	Seq	Rnd	Seq
Messenger	73.7%	26.3%	90.7%	9.3%	63.2%	35.7%
Email	52.8%	45.2%	84.5%	13.7%	61.6%	33.7%
Search	49.8%	45.1%	87.7%	8.5%	70.9%	24.5%
UserContent	58.3%	39.4%	93.1%	5.5%	73.2%	25.0%
D-Process	30.1%	68.8%	73.2%	26.8%	45.4%	54.4%
DisplayAds	96.5%	2.5%	93.5%	4.3%	78.5%	19.2%
TPCC	68.8%	31.2%	97.2%	2.8%	71.1%	29.9%
TPCE	91.3%	8.7%	91.9%	8.2%	77.7%	22.4%
TPCH	96.7%	3.3%	65.5%	35.5%	52.8%	47.2%
Exchange	32.0%	68.1%	83.2%	16.8%	68.1%	31.9%

Table A.8: Random/Sequential characteristics before and after defragmentation.

Estimating performance after defragmentation, using the models, can act as an offline method to identify the benefits of defragmentation, as well as the optimal moment to perform defragmentation, without having to examine the entire application. These are usually latency-critical applications that cannot afford the overhead of a continuous online evaluation.

In most cases, the percentage of sequential accesses increases by 20% (Table A.8), which corresponds to a storage speedup of 8-45% as shown in Figure A.9. This result implies that clustering I/Os is more beneficial than taking advantage of parallel spindles, due to faster completion of sequential accesses. Two applications that benefit from this are D-Process and Email, which have the highest write-to-read probabilities. Since these applications are random-write dominated, improving their sequential characteristics allows better utilization of the full disk rotation. Defragmentation also benefits the TPC benchmarks that access continuous entries in database tables.

A comparison between the benefits of SSD caching and defragmentation shows that for some workloads (Email and Display Ads) defragmentation offers better speedups without increasing the cost of the system. The decision on which method is most beneficial at a certain time is up to the storage system designer, given the application and system of interest.

A.6 Conclusions

In this work we have proposed a framework for modeling, characterization and re-generation of large-scale storage workloads. We have extended a probabilistic model to capture granular information on a workload’s I/O pattern and implemented a tool that recreates DC workloads with high fidelity. This tool can be used for a wide spectrum of studies for large-scale systems, without the need to access DC application code or full application deployment.

In contrast with previous work, we take into account spatial and temporal locality of I/O accesses, a critical feature for DC applications. We have conducted detailed characterization of the storage activity of DC applications and performed extensive validation of the generated I/O traces against ten real workloads. Finally, we have evaluated two possible uses for the tool, SSD caching and defragmentation, and quantified the improvement in performance. We believe that, compared to previously available workload generators, this framework can be used to make confident design decisions for DC systems.

Appendix B

BLOC: Bandwidth-Aware Storage Consolidation

B.1 Introduction

Consolidation can significantly improve datacenter (DC) *resource efficiency*, by increasing server utilization by integer factors. First, consolidation can pack hosted workloads onto fewer servers, significantly increasing DC utilization, which currently ranges between 5% and 15% [29, 31]. Second, aside from reducing the number of servers required, consolidation also reduces energy consumption per unit work. This happens because servers are not energy-proportional and consume significant energy even when idling [31, 99, 191]. Consolidation amortizes this idle power over more units of work.

Consolidation methods based on virtualization for compute-intensive workloads are now available both in commercial products [130, 2, 4] and through cloud platforms including Amazon EC2 [13] and Windows Azure [279]. Sophisticated consolidation methods for network-intensive workloads are also available [54, 55, 61, 183]. However, while several storage virtualization mechanisms exist [130, 145, 181, 243], performance-aware consolidation policies are lacking.

Many storage consolidation policies have considered capacity requirements [126, 140, 141, 194]. Although this ensures that consolidation is feasible, it does not ensure

that performance will not be penalized. Storage performance is very tightly coupled to bandwidth, and surges in bandwidth usage result in significant performance losses. Thus bandwidth must be taken into account to preserve performance. Additionally, while storage capacity costs have dropped rapidly [163], bandwidth costs have not. Therefore, it is important to use bandwidth efficiently. There is work on storage system design exploration [18] which accounts for bandwidth requirements to find the optimal design for a given workload, but does not determine the set of applications to be consolidated and assumes that the storage system of each machine is configurable, which is not the case in a homogeneous DC.

A second challenge is that *bandwidth requirements are highly dynamic* [31, 85, 258]. Consider storage-intensive applications serving user data, such as pictures on a social network or user mailboxes on an email server. More users are active at certain times of the day with many data objects being simultaneously accessed, resulting in high storage bandwidth use. At other times, far fewer objects are accessed. Provisioning for peak demand wastes energy at off-peak times, since individual nodes are not energy-proportional. Instead, we want a storage system that adapts to fluctuations in user demand. However, changing the number of active nodes, and the bandwidth allocated to each application is challenging, since all data must remain accessible. If some nodes are to be powered down or re-allocated to an application with increasing demand, their data may need to be moved.

Finally, when the datasets of several applications are consolidated on the same storage node they interfere in shared storage resources, i.e., bandwidth and capacity. Detecting which applications are sensitive to interference is critical to determining which datasets can be consolidated effectively. Interference in storage resources depends on the specific access pattern of a workload, however exhaustively evaluating all possible consolidation combinations to determine the ones that do not degrade performance would be infeasible in an environment with strict QoS requirements.

In this chapter we present BLOC, a storage consolidation scheme that addresses these challenges. BLOC detects interference between applications, and consolidates workloads to the *extent that performance is not penalized*, while *adapting to dynamic bandwidth demands*, much like VM consolidation methods tune the number of active

servers to match compute demand [35, 165, 175, 273]. We address the first challenge by explicitly limiting the bandwidth usage on each storage node to the level that preserves the performance requirements of each consolidated application. We characterize application bandwidth usage to determine this limit. To address the second challenge, we dynamically tune the number of active nodes to match user demand, based on expected future load. In this case, data migration may be required to ensure data availability. BLOC predicts near-future application demand and starts data movement ahead of time. It reserves extra bandwidth for this proactive migration, such that as utilization grows from migration, it does not grow against the consolidated applications' bandwidth. Finally, to minimize storage interference between consolidated applications, we use analytical models to create mirror instances of each storage workload. These models capture the application's temporal and spatial I/O patterns, but enable much fast evaluation of how sensitive application mixes are to storage interference, than using the original workloads. The end result is a more energy-proportional storage subsystem although individual storage nodes are not energy-proportional.

Prior work has addressed the problem of tuning system bandwidth to match demand by adjusting the number of active replicas for each object. When the required bandwidth for a data chunk goes down, some of its replicas can be powered-down, and systems such as Sierra [258] and Rabbit [14] do precisely that, while respecting reliability requirements. However, bandwidth decrease may not always be due to smaller demand for a given data chunk (resulting in a need for fewer replicas). When fewer users are accessing their pictures or emails, fewer data chunks are being accessed. Existing systems do not reduce the provisioned bandwidth or the number of storage nodes. We describe our methods assuming a replication factor of one and later discuss how previously-proposed techniques can be integrated to tune the replication factor. Additionally, in distributed file systems, including GFS [?] that now use Reed-Solomon codes for fault-tolerance instead of replication, reducing replication is not feasible to reduce power consumption.

The main contributions of this chapter are three:

- **Interference-aware storage consolidation:** We develop a practical technique

to quickly characterize the sensitivity of different application mixes to storage interference. The technique relies on validated analytical models that preserve workload I/O access patterns, and allows interference characterization that would take several days to complete within minutes. We use this characterization to determine an efficient packing of application datasets on storage nodes that does not violate their performance constraints.

- **Demand adaptation:** We design a policy to dynamically reallocate storage resources among applications such that the number of active nodes matches user demand. Extra bandwidth is reserved to overcome migration overheads. We pre-activate additional nodes and transfer data to them based on near-future prediction of when an increase in node allocation will be needed.
- **Validation with real DC workloads:** We evaluate BLOC using I/O traces of 7 large-scale Microsoft DC applications and 3 TPC benchmarks on a 40-machine cluster. BLOC achieves 3.09x energy savings over a system without storage consolidation. We also compare BLOC against a capacity-based scheme. While the capacity-based scheme reduces energy use, it degrades both throughput and latency by almost 2x. BLOC maintains performance within 2.8% compared to performance before consolidation while yielding significantly higher energy savings than the capacity-based scheme. Finally, we demonstrate how BLOC can be tuned to trade off performance for even higher energy savings.

B.2 BLOC Design

B.2.1 System Overview

We design BLOC with the principle of accounting for both the performance and energy impact of storage consolidation, by minimizing storage interference and predictively plan for changes in user demand. For the evaluation of BLOC we use applications that rely on a distributed file system (DFS), therefore all BLOC techniques are presented under the assumption of an underlying DFS. With appropriate

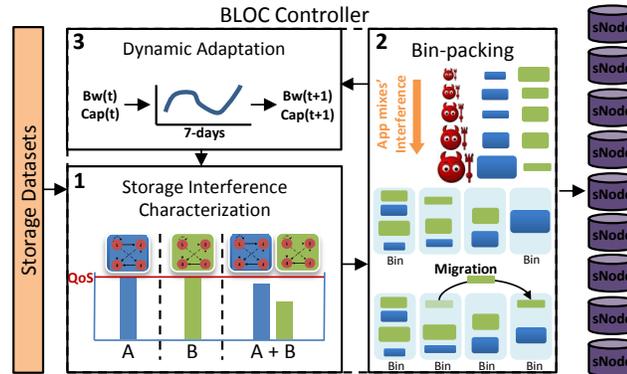


Figure B.1: BLOC architecture.

changes, the principles in BLOC can be applied in systems such as VM management systems, RDBMS and scalable key-value stores as well (see Section B.8).

Figure B.1 shows the primary components of BLOC. *Storage datasets* represent the backend of workloads that are being consolidated. The *storage nodes* represent the servers where the workloads are hosted. Each storage node has a given number of disks and a disk array controller configuration¹ that determine its maximum storage capacity and bandwidth. The actual capacity and bandwidth that can be used by an application depend on the workload’s I/O access pattern.

BLOC sits in the middle and consolidates applications in storage nodes, with the objective of improving utilization without hurting performance. The first step in BLOC performs *characterization of storage interference*, which addresses the issue that depending on applications’ resource demands and access patterns, performance can suffer from consolidation, due to interference in the storage subsystem. This step determines whether a mix of applications can be effectively consolidated, and what is the expected performance degradation - if any - after consolidation, due to interference.

The second step performs the *bin packing* of application storage into fewer storage nodes. The algorithm accounts for the sensitivity to interference and the resource demands of the different consolidation candidates to determine the per-application

¹Both Redundant Array of Inexpensive Disks (RAID) and Just a Bunch of Disks (JBOD) configurations are used in commercial DCs.

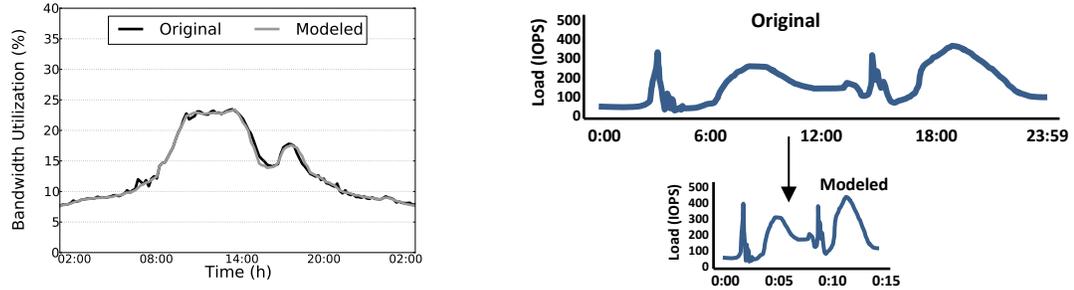


Figure B.2: Validation of the storage activity model for Websearch (Figure B.2a) and compression of storage load using the analytical model (Figure B.2b).

resource allocation.

Finally, the third step of BLOC accounts for *changes in user load over time*. This component characterizes the per-application temporal demand variation and predicts future resource requirements. This information is used to re-evaluate storage interference and is then given to the bin packing component to adjust the allocation of resources, if necessary. This component also compensates for prediction errors, attempts to minimize data migration and manages the excess resources required if migration becomes necessary.

The following three sections describe each component in detail.

B.3 Quantifying Storage Interference

B.3.1 Motivation

Interference is the result of contention in a shared resource. Consolidating applications in the same storage node can introduce contention in the storage subsystem, resulting in performance losses. Therefore quantifying the degree of interference between two or more consolidation candidate workloads is critical to ensure that any efficiency gains from consolidation do not come at a performance penalty. Ideally sensitivity to interference would be determined by profiling consolidated applications and measuring the performance losses from contention. Obviously exhaustive profiling of all possible application mixes in a DC environment is infeasible. Additionally,

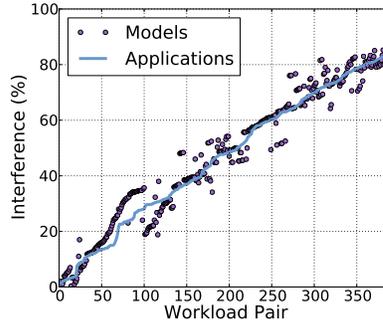


Figure B.3: Validation of the interference estimation using the analytical workload models across the surviving 390 application pairs. Deviation between estimated and measured interference is on average 3.8%.

interference depends on load variations, therefore profiling would need to happen over long time windows to cover the whole spectrum of user activity. Finally, profiling application mixes has system side-effects as it requires migrating large datasets and allowing the applications to potentially change the datasets' state.

B.3.2 Application Models

We address the challenges above by leveraging the analytical storage models described in [85]. This model captures block-level I/O patterns in a concise way using Markov Chains (MC) and allows us to recreate time-varying storage activity, that preserves the timing and locality of the original I/O accesses. It also enables varying the load intensity to create multiple demand levels. Models are retrained over time to capture changes in application code and/or structure. While these models were validated in [85], we validate them against the storage traces of our workloads. Figure B.2a shows the storage bandwidth for a large-scale deployment of Websearch, running on 1,000 servers, over a 24h period (black line). The grey line shows the storage activity generated using the model in a scaled-down system with 40 servers. The synthetic workload follows the original activity fluctuation with average deviation 3.8% and similar per-hour deviations. We use this storage model for performance measurements, both to characterize interference and to evaluate the impact of consolidation.

Additionally, due to its statistical nature, the model enables compressing the

storage activity of applications in time to evaluate more quickly a wider spectrum of user loads. Figure B.2b shows how the model captures the full storage behavior of a workload in a shorter timescale. While the trace from the original application is recorded over a 24 hour period, the model is generating the same load fluctuation over 15 minutes. This happens by switching between MCs at a faster pace than the original workload [85]. Note that this does not affect the representativeness of the I/O patterns. The inter-arrival times between requests remain the same as in the original workload; what changes is the time for which the application remains at a specific load, e.g., if the original workload experienced a 30 minute interval with a flat load of 100 IOPS, this interval is reduced to 30 seconds when using the model. The exact degree of compression depends on the application patterns, however in general, a week’s worth of storage traces can be compressed to a few hours using the analytical model.

B.3.3 Trimming the Search Space

This still leaves the problem of the exponentially increasing space of application mixes. If BLOC were to evaluate interference naïvely for all consolidation pairs of N applications, it would have to assess 2^N combinations. This number increases further for mixes of more than 2 workloads. Obviously, the overhead of this process would be infeasible. In BLOC we trim the search space, by discarding from consideration application mixes that are bound to experience performance violations when consolidated. These are mixes for which the aggregate storage bandwidth or capacity exceeds what the storage node can support. Note that the bandwidth the storage node can support in general, is different from its rated sequential bandwidth, and depends on the bandwidth that the most random application in the mix can sustain. For the workload combinations that remain, BLOC quantifies storage interference.

B.3.4 Profiling

For each active application in the system, BLOC creates a “mirror” instance using the analytical model. This mirror instance is used to quantify the degree of interference

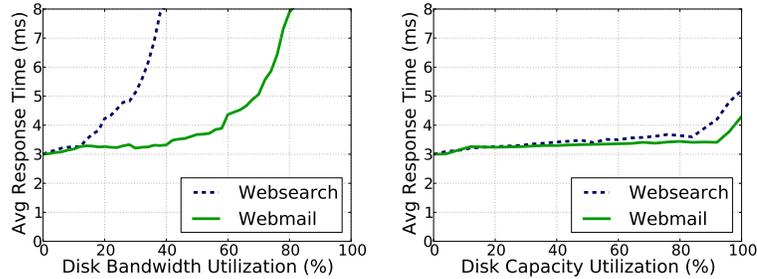


Figure B.4: Performance variation with bandwidth and capacity utilization.

between consolidation candidates. While BLOC allows consolidation of *any number of workloads*, the initial step only considers application pairs, since they are more likely to be consolidated without QoS violations. Each of the application pairs that did not get discarded in the previous step is profiled for a 1-5 minute period, while the controller evaluates the performance degradation for each workload. The metric we use for the degree of interference is the average performance degradation across the pair’s workloads compared to running in isolation. Higher interference means higher performance degradation. Note, that even after trimming the application space, the total number of pairs that have to be examined is high. However, the full evaluation across pairs only needs to happen at the initial step of consolidation, when no application has been consolidated. This step can happen offline. When the system is online, the only mixed of applications that need to be examined are those for which consolidation has to change, i.e., some datasets have to be migrated. This number is significantly smaller than the complete application mix space.

Since the model preserves I/O access locality and timing, the interference determined using the workload models is expected to be very close to the interference that the actual applications will experience. In the following section we validate that this is indeed the case.

B.3.5 Validation

We use 10 production DC workloads from Microsoft for our evaluation. All applications run over a DFS. Section B.7 provides details on each application. The full space

of application pairs consists of $2^{10} = 1024$ pairs (note that the number of pairs is not simply $\binom{N}{2}$, since the order in which applications are consolidated matters). From these, 390 survive the maximum bandwidth and capacity check. Figure B.3 shows the predicted versus measured interference over the 390 surviving application pairs. Pairs are ordered from those experiencing the *least* to those experiencing the *most* interference.

Interference ranges from 0 to 85%, which means that performance degradation for application pairs varies from marginal to very serious. The deviation between predicted and measured interference is on average 3.8% and at most 11.8%. Using the analytical models enables both fast and accurate interference estimation, without the overhead of deploying each real workload multiple times. Also in general the mirror instances overestimate interference, which makes the scheme slightly more conservative than necessary, but favors preserving QoS.

B.4 Bin Packing

Given the application resource requirements and the degree of interference for different application pairs, storage consolidation can be viewed as a bin-packing problem. The application datasets are the objects to be packed. Since each object has size requirements in both storage capacity and bandwidth, this is a two-dimensional bin-packing problem. Theoretically, the size of each bucket is the total capacity and bandwidth of the storage node. In practice these limits cannot be met, as described in the following section.

B.4.1 Limiting Storage Resource Usage

As utilization increases the response times the storage system can provide increase. Figure B.4a shows this effect for two distributed applications (Websearch and Webmail) with varying bandwidth utilization. Both applications run on production clusters with thousands of servers. The utilization on the x-axis is normalized to the rated storage bandwidth of the node. Suppose the response-time constraint for each

application is 5ms. Then, for Websearch, the maximum supported bandwidth before latency exceeds 5ms is 30% of rated bandwidth, while for Webmail, it is 70%. This is expected since Websearch has a more random access pattern than Webmail.

We also perform a similar measurement for capacity. Figure B.4b shows that performance changes with capacity utilization, especially at higher utilizations - although less than it changes with bandwidth. This is likely due to increased fragmentation and increased seek times from traversing more tracks. To ensure performance, we enforce a similar constraint on capacity.

The maximum bandwidth and capacity limits that a storage node can support for an application i are denoted MB_i and MC_i . These values need to be recalculated after major application software updates or node configuration changes that may cause I/O access patterns to change.

B.4.2 Bin Packing Algorithm

First, we determine the maximum bandwidth and capacity an application i can use on a storage node, MB_i and MC_i , as described in Section B.4.1. This is used to determine the total bandwidth limit, MB_s , of a storage node, s , after applications $i = \{1, \dots, N\}$ are placed on it:

$$MB_s = \min_{i=1\dots N} MB_i \quad (\text{B.1})$$

The capacity limit, MC_s is calculated similarly.

Since we want to preserve the per-application performance constraints, BLOC first packs applications that interfere minimally in the storage system.

We start with a system where applications are not consolidated. All application pairs that survived the maximum capacity/bandwidth check are evaluated and sorted in order of increasing interference. Applications are removed from the sorted list and packed on storage nodes until either the capacity or bandwidth limits of the node are met, or interference exceeds a tolerable threshold. For the sake of explanation, we assume that the highest performance degradation across the workloads of a mix that can be tolerated is 10%. Every time a new application is placed on a server, the

node's available storage resources (Expression B.1) and level of interference ($Interf_s$) change. This process is repeated until all available applications have been assigned to some node. The available storage bandwidth also accounts for a small Δ_B for migration (Section B.5.3). The details are specified in Algorithm 1.

Algorithm 1 CONSOLIDATE($\mathcal{I}, \mathcal{K}, \mathcal{B}, \mathcal{C}, \Delta_B$)

Require: \mathcal{I} : applications, \mathcal{B}, \mathcal{C} : b_i and c_i are the bandwidth and capacity requirements respectively of application i , \mathcal{K} : available nodes

Ensure: Placement P , that assigns all applications in \mathcal{I} to some node in \mathcal{K}

```

1: for  $i, j$  in  $\mathcal{I}$  do
2:    $Interf_{(i,j)} = avgPerfDegradation_{(i,j)}$ 
3: end for
4:  $\mathcal{I} \leftarrow$  application pairs in  $\mathcal{I}$  sorted by increasing  $Interf_{(i,j)}$ 
5: Use storage node  $s = 1$   $\triangleright$  nodes taken from  $\mathcal{K}$ 
6: Initialize  $MB_s = \infty, MC_s = \infty, B_s = 0, C_s = 0$ 
7: for  $i = 1$  to  $|\mathcal{I}|$  do
8:    $x \leftarrow$  Is  $B_s + b_i < \min(MB_s, MB_i) - \Delta_B$ ?
9:    $y \leftarrow$  Is  $C_s + c_i < \min(MC_s, MC_i)$  ?
10:   $z \leftarrow$  Is  $Interf_{(s)}$   $\downarrow$  10% ?
11:  if  $x$  is false OR  $y$  is false OR  $z$  is false then
12:     $s \leftarrow s + 1$ 
13:    Initialize  $MB_s, MC_s = \infty; B_s, C_s = 0$ 
14:  end if
15:  Assign  $i$  to  $s$ 
16:   $B_s = B_s + b_i, C_s = C_s + c_i, Interf_{(s)} = Interf_{(s-i)} + Interf_{(s,i)}$ 
17:   $MB_s = \min(MB_s, MB_i)$ 
18:   $MC_s = \min(MC_s, MC_i)$ 
19: end for

```

B.4.3 Consolidation Granularity

In the description above, we assumed that each application to be consolidated is the dataset hosted by a workload in a single node. However, this granularity is very limiting, since consolidation can only be done for entire node instances, reducing the potential efficiency savings. The granularity at which consolidation can be performed is a function of the system applications operate on, e.g., DFS, key-value store, etc.

Since our applications operate over DFS, the dataset of a node can be split into partitions at file granularity, which are then used as the consolidation units. The split is done using the same storage model as before, to preserve I/O spatial locality. Typically, a node’s dataset is split to 16-64 subsets.

This increases the number of objects being packed, leading to higher server utilization. The bin packing algorithm has low computational complexity and easily scales to the larger number of objects. The technique to determine the maximum bandwidth and capacity, MB_i and MC_i , still applies in a similar way to each of the individual storage subsets.

B.4.4 Dynamic Demands

The bin packing described above is a static algorithm and does not account for changes between time intervals. While it can be naïvely re-applied at each time interval, that is inefficient, since it may result in excessive data migration. Rather, we update the placement incrementally, based on the change in application demands.

Even with incremental updates, the number of active nodes changes from one interval to the next. When the number of nodes decreases, the extra nodes may be turned off any time after the interval required for data movement. However, when the number of nodes is increasing, additional nodes should be brought online sufficiently in advance such that data can be moved to them before demand increases.

We predict the resource requirements for the next interval to determine which nodes are expected to exceed their capacity and/or bandwidth limits. We then select the applications with the smallest data footprints on these nodes and compute a new placement only for them, since they are the only workloads that need to be moved. Any extra nodes required for the change in placement are brought online. Section B.5.2 describes how the dynamic phase of bin packing works.

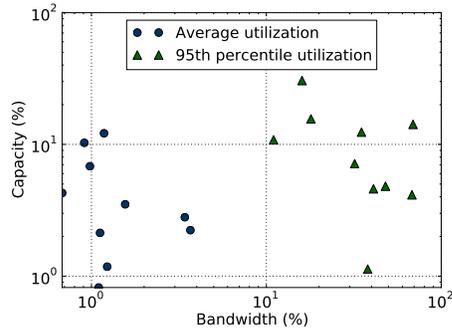


Figure B.5: Utilization of storage bandwidth and capacity across the 10 DC workloads. Log scale used to show the low utilization region clearly.

B.5 Dynamic Adaptation

Most applications go through different phases during their execution. This is especially true for interactive DC workloads, whose load is a function of user activity [31, 195, 258, 163, 85, 84, 88]. Consolidation should adapt to these changes to avoid performance degradations. This means that there may be periods that require certain datasets to be migrated to avoid oversubscribing storage resources in consolidated servers.

To avoid the overheads of migration, the controller could naïvely provision for peak storage resource requirements. However, for the majority of cases this is extremely wasteful. Figure B.5 shows the difference between average and 95th percentile storage resource utilization for the ten examined workloads. Storage bandwidth and capacity are normalized to the values the server can support for each application, since depending on their access patterns, different workloads get very different maximum bandwidths on the same storage node. The key observation here is that average utilization rarely exceeds single digits, while 95th percentile utilizations are significantly higher.

Instead of conservatively provisioning for peak load, in BLOC we provision for current storage resource requirements, but prevent oversubscription with two mechanisms; first, we set a maximum limit for storage capacity and bandwidth utilization as described in Section B.4.1. This prevents the system from constantly operating

in a close-to-saturation region and in the case of unexpected load spikes it lets demands grow against the unused resources, as opposed to interfering with application execution.

Second, to avoid migrating data when a surge in load arrives, the system acts proactively, and starts the migration before the high load arrives. This way, when user demand increases, all storage resources are dedicated to servicing user requests, as opposed to migrating data.

B.5.1 Predicting User Load

Load may be classified into two categories: primary user load and spikes. Primary user load follows a diurnal pattern [31, 85, 84, 195, 258] and can be predicted within a reasonable margin of error. Spikes, on the other hand, occur randomly and are hard to predict [31, 99]. We use traced behavior to predict the primary user load and adapt to spikes and other deviations at run time. The primary user load has large fluctuations and adapting to its variation can yield significant savings.

The implication of Figure B.5 is that although provisioning for peak demands is wasteful, each application does not come with a specification of user demand and bandwidth requirements over time. This must be predicted at run time. The prediction method in BLOC is based on preceding user behavior. To compensate for deviation from past behavior, we include an additional runtime correction term.

Predictions are performed at a 30-minute granularity. Changing resource allocation at shorter intervals can improve the match between allocated resources and demand, but each update comes at the cost of potential data migration. Hence, updates should not happen faster than data movement can be amortized. For the studied applications, 30 minutes are sufficient to track major load changes and perform any migration required (see Section B.7.2).

The outputs of the demand prediction module are the bandwidth and capacity requirements expected in the next time interval. If $b_i(t)$ and $c_i(t)$ are the bandwidth and capacity requirements of application i at time interval t , then this module outputs $b_i(t + 1)$ and $c_i(t + 1)$. The actual implementation is given in Section B.6.

B.5.2 Dynamic Updates to Bin Packing

This phase is described in Algorithm 2. At each time interval, the predicted resource demands are used to determine the nodes for which any application will violate its resource constraints in the *next* interval, due to increased demand (Steps 5,6). To minimize the amount of data that needs to be migrated, applications with the smallest data footprints are removed from these nodes and added to the set of applications, \mathcal{U}_A that will be reallocated (Steps 7 to 15), until the node utilization drops below the allowed limit. Nodes where resource demand falls below the limit minus the hysteresis margins δ_B, δ_C , either due to reduction in the demands of their current applications, or due to deconsolidated workloads, are added to the set of nodes available for consolidation (Steps 16, 21). These nodes are partially occupied but the remaining capacity is available for consolidation. The static consolidation algorithm operates on this new, reduced application set \mathcal{U}_A , instead of performing a global re-assignment. For the portion of the placement that changes, data movement is started within the current time interval, so that all storage resources are used towards servicing user requests in next time interval². If additional servers are required by Algorithm 2 they are turned on, and any servers relieved of all applications are turned off.

The incremental update used in the dynamic phase does not consider all applications and may deviate from a bin packing solution that examines all applications in each interval. Thus, a global re-assignment may be performed periodically, such as once a day, during times of low demand. This can also help ensure that the same nodes are not repeatedly power-cycled by randomizing the order in which nodes are assigned.

B.5.3 Migration Costs

Data movement is not free. It incurs two types of costs: *performance*, since some of the system bandwidth will get tied up in migration, and *energy*, since the energy consumption of extra I/Os performed for migration will eat into the savings from

²We assume that each data block is unavailable during migration. It can be accessed before movement starts or after it has been completely copied to the new location. The movement time for each individual block is small.

Algorithm 2 UPDATEPLACEMENT($P(t)$, $\mathcal{B}(t+1)$, $\mathcal{C}(t+1)$)

Require: $\mathcal{B}(t+1)$ and $\mathcal{C}(t+1)$ specify predicted bandwidth and capacity requirements, respectively, for all applications at next time step

Ensure: Updated placement that satisfies new demands

- 1: Compute bandwidth and capacity utilizations, $B_s(t+1)$ and $C_s(t+1)$, at each node $s \in \mathcal{K}$, using demands $\mathcal{B}(t+1)$, $\mathcal{C}(t+1)$ and current placement
 - 2: $\mathcal{U}_A \leftarrow \{\}$ ▷ set of applications to update
 - 3: $\mathcal{U}_K \leftarrow \{\}$ ▷ set of storage nodes made available
 - 4: **for** s in \mathcal{K} **do**
 - 5: $x1 \leftarrow \text{Is } B_s(t+1) > MB_s ?$
 - 6: $y1 \leftarrow \text{Is } C_s(t+1) > MC_s ?$
 - 7: **while** $x1$ or $y1$ **do**
 - 8: $k \leftarrow$ application with least data on s
 - 9: $\mathcal{U}_A \leftarrow \mathcal{U}_A \cup k$
 - 10: $B_s(t+1) \leftarrow B_s(t+1) - b_k(t+1)$
 - 11: $C_s(t+1) \leftarrow C_s(t+1) - c_k(t+1)$
 - 12: $Interf_s(t+1) \leftarrow Interf_s(t+1) - Interf_{(s,i)}(t+1)$
 - 13: Update MB_s, MC_s to value with k removed
 - 14: Update x, y using new $B_s(t+1), C_s(t+1)$
 - 15: **end while**
 - 16: $x2 \leftarrow \text{Is } B_s(t+1) < MB_s(t+1) - \delta_B ?$
 - 17: $y2 \leftarrow \text{Is } C_s(t+1) < MC_s(t+1) - \delta_C ?$
 - 18: $z2 \leftarrow \text{Is } Interf_s(t+1) < 10\% ?$
 - 19: **if** $x2$ or $y2$ or $z2$ **then**
 - 20: $\mathcal{U}_S \leftarrow \mathcal{U}_S \cup s$
 - 21: **end if**
 - 22: **end for**
 - 23: $P(t+1) = \text{CONSOLIDATE}(\mathcal{U}_A, \mathcal{U}_K + \text{empty nodes}, \mathcal{B}(t+1), \mathcal{C}(t+1), Interf_{(i,j)}, \Delta_B)$
 - 24: Migrate data of nodes that change from $P(t)$ to $P(t+1)$
-

powering down unused nodes.

Performance: To mitigate the performance impact, BLOC uses two mechanisms. First, data migration is performed *ahead of load increase*, based on predicted demand. This way upon arrival of high load, storage resources are *explicitly* used to service user requests. Second, BLOC maintains an *additional bandwidth margin*, Δ_B , beyond what is required to serve the workload demand. This implies that workloads are

packed slightly sparser than allowed based on their bandwidth requirements. This way, if the server is operating near its limits, migration uses the reserved bandwidth and does not take bandwidth away from the consolidated applications.

The value of Δ_B should be sufficient for all data movement required by applications on a server. We empirically determine a common value for Δ_B at 2% of rated bandwidth on all nodes, based on the load changes between intervals and the corresponding application re-allocations.

Energy: To reduce the energy overhead of migrations, data should only be moved when the savings from a reduced number of active nodes will be larger than the migration energy cost. This is accomplished by using a hysteresis margin below the bandwidth and capacity limits. If resource usage in some node falls, but is within the hysteresis margin (i.e., not enough to justify migration), applications are not re-consolidated to fewer nodes.

Most modern DCs use mechanical hard disks. The idle cost of keeping a storage node active with disks spinning is much larger than the energy cost of a small migration bandwidth. This implies that small hysteresis margins, δ_B and δ_C for bandwidth and capacity respectively, can be set. These values are common across all nodes regardless of application placement. Since the bin-packing procedure does not pack each node to its limit, in practice this margin does not lead to wasted resources. δ_B and δ_C may need to be tuned more precisely for solid state storage.

B.5.4 Performance and Efficiency Trade-offs

The design of BLOC presented so far is geared towards preserving performance. In some scenarios, such as in batch-processing workloads, resource efficiency may be more important than performance. In this case, BLOC can be tuned to provide further savings for selected applications. Parameters MB_i and MC_i can be tuned to higher values for applications that prioritize efficiency over performance. We evaluate the trade-off between efficiency and performance in detail in Section B.7.2.

Second, we can reduce the fraction Δ_B of excess resources that covers migration overheads, to improve overall resource efficiency. This implies that whenever data

movement is performed, performance may briefly suffer.

Third, deconsolidation can be performed lazily. Rather than deconsolidating when an increase in demand is predicted, the number of nodes are only increased when an actual increase in demand is observed. This saves resources in the event of an erroneously-predicted increase in demand. However when demand does increase, there will be a performance penalty due to the delay in bringing up additional nodes.

We experimentally evaluate the savings achieved through some of these modifications and their performance impact. The results can be used to tune the system for the best trade-off between performance and efficiency.

B.6 Implementation

State: BLOC resides with the cluster scheduler and determines the allocation of storage nodes to applications. The implementation of the methods presented earlier requires maintaining some state for each application object (either entire application instance or data subset) and each storage node:

- *Per-application state:* BLOC tracks each application’s historic and real-time capacity and bandwidth requirements and its interference against other workloads. It also determines the maximum bandwidth (MB_i) and capacity (MC_i) limits for each application.
- *Per-server state:* For each storage node, BLOC records the current capacity and bandwidth utilizations (C_s, B_s), the set of applications that are hosted on it, its current interference levels, and the current maximum capacity and bandwidth limits of the server (MB_s, MC_s) based on the hosted applications.

The state overhead is minimal, in the order of a few KBs, even for a DC with a large number of applications. This state information is used by the algorithm described in Section B.4 to consolidate applications into fewer nodes.

Prediction: Another implementation detail is the specific prediction method used to track application demand dynamics. Our prediction includes two terms. The first term exploits the application’s past behavior. At a given time interval the expected

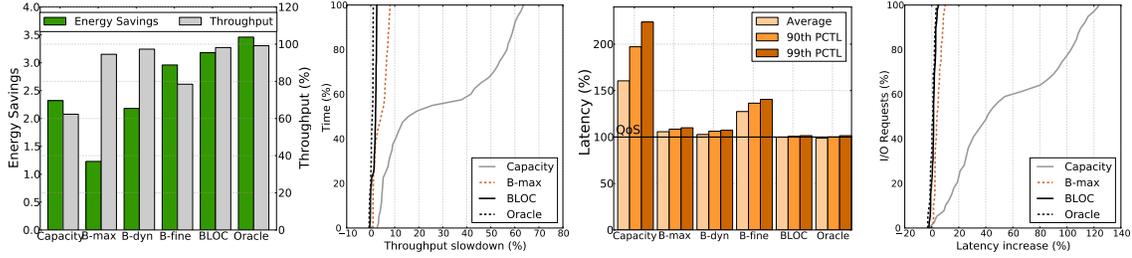


Figure B.6: Performance and efficiency comparison between different consolidation schemes.

demand is predicted by averaging the resource (bandwidth or capacity) use over the past seven days for the same time of day, similar to [258]. This information is sufficient since most applications change slowly and the algorithm has some resource slack. This history-based predicted demand is denoted $R_h(t + 1)$. The second term accounts for deviations from past behavior using the real-time observation, $R_{rt}(t)$. The two terms are combined to generate the final prediction as follows:

$$R(t + 1) = \begin{cases} (1 - A^2) \cdot R_h(t + 1) + A^2 \cdot R_{rt}(t) & \text{if } A < 0.3 \\ (1 - A) \cdot R_h(t + 1) + A \cdot R_{rt}(t) & \text{if } 0.3 \leq A < 0.7 \\ R_{rt}(t) & \text{if } A \geq 0.7 \end{cases}$$

Weight A is defined as the ratio of the number of intervals where observed demand deviated from predicted demand by more than 5%, over the total number of intervals for which prediction has been running. The more demand deviates from past behavior, the greater the value of A , which biases the prediction towards current observations. The values 0.3 and 0.7 that define the weight of the past versus the real time information in the estimation of R are determined based on the standard deviation of $(R_h - R_{rt})$, similarly to the technique discussed in [247].

Operation: Let’s assume that the system starts with a set of applications running without storage consolidation. First, the storage activity of each application i is modeled to determine MB_i and MC_i . Then, consolidation is enabled and applications are ranked according to their degree of interference. Bin-packing starts using Algorithm 1 and applications get placed on the minimum required number of nodes. BLOC then

prepares for the next interval using Algorithm 2, which runs at the chosen time granularity of 30 minutes. Algorithm 2 selects the subset of applications that will need to be re-allocated to accommodate changes in user load.

B.7 Evaluation

We have deployed BLOC on a 40-machine cluster and used it for application storage consolidation. We compare it against an existing consolidation approach that only considers capacity requirements (**Capacity**), as well as a hypothetical scheme that has perfect future knowledge (no prediction errors) and incurs no migration costs (**Oracle**). Computing the oracle placement is an NP-hard problem and we use an exhaustive search for our purpose.

We also compare four different configurations of BLOC to explicitly show the gains from each of the design techniques. These are (1) **B-max**, a static scheme that only uses the maximum bandwidth (MB_i) determined using BLOC for each application but does not adapt to dynamic demand. B-max only uses algorithm 1 once. (2) **B-dyn**, a dynamic scheme that accounts for demand variation but does not split application storage to smaller fractions, i.e., it treats the dataset of an application in a node as a single application object. B-dyn makes use of both algorithms 1 and 2. (3) **B-fine**, a scheme that performs fine-grained consolidation, but makes static consolidation decisions, i.e., it only accounts for initial demands without adjusting to dynamics. B-fine only uses algorithm 1. Finally, (4) **BLOC**, the final design that incorporates all of the previous techniques.

I/O Workloads: We use storage traces from production clusters of seven large Microsoft applications: Websearch, Webmail, D-Process (a distributed computing application similar to MapReduce [69]), Messenger, User Content (a cloud storage service), Display Ads, and Exchange. Websearch, Webmail, Messenger and Exchange are user-interactive applications that experience significant load variations throughout a day, while applications like D-Process do not depend on external user traffic. These applications run over a DFS and span thousands of servers. We also use I/O access traces collected from large installations of three benchmarks from the TPC suite:

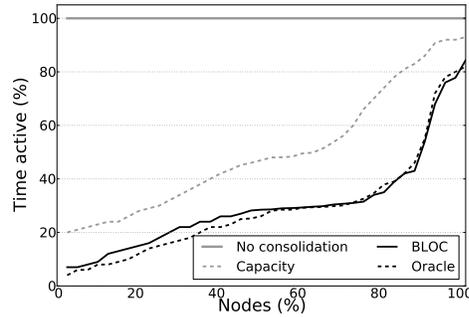


Figure B.7: CDF of storage node active time.

TPC-C, TPC-E and TPC-H.

Testbed: We use a 40-machine cluster with three server types. The first type has 2 sockets, 12 cores, 12GB of DRAM and 4 hard drives (1.2TB). The second has 2 sockets, 24 cores, 48GB of DRAM and 4 hard drives (1.2TB). The third type has 2 sockets, 8 cores, 8GB of DRAM and 12 hard drives (5.4TB). All hard drives are 10krpm HDDs organized in RAID1+0 configurations. The cluster has 18 servers of the first type, 10 of the second and 12 of the third. We determine maximum bandwidth and capacity, MB_i and MC_i , for every application on each type.

Each of the 10 applications has between 3 and 5 instances for a total of 40 instances on the cluster. Instance data sizes range between 50GB and 450GB. For B-fine and BLOC, where data is split into smaller chunks, the consolidated objects are between 782MB and 1963MB. The I/O traces are replayed such that the same block is accessed for a given I/O operation, whether all accesses of an application go to the same node or are split across nodes. Addressing issues on privacy between applications are out of the scope of the current work. BLOC may place data from different applications in the same storage node. All experiments are run for 24h periods and repeated seven times for consistency.

B.7.1 Comparison of Consolidation Schemes

Performance and efficiency: Figure B.6a compares the energy savings across the six schemes. Savings are defined as the total amount of energy spent for the cluster, compared to energy spent in the same cluster, for the same number of requests,

without consolidation (1 on the energy savings axis represents no savings while 2 represents a 2x reduction in energy consumption). These savings are cluster-wide and account for data migration overheads. The savings account for the increase in energy consumption due to higher utilization after consolidation, as well as the reduction in cluster-wide consumption due to unused nodes being powered down.

Resource efficiency alone is not a sufficient figure of merit, since consolidation can incur performance losses. Figure B.6a also shows the performance in throughput, including any overheads from migration. Latency is shown in Figure B.6c, normalized to the latency of applications running in isolation, which is represented as 100% on the y-axis.

While the capacity-based scheme provides significant savings, it degrades throughput by nearly 36.8%. B-max considers bandwidth explicitly and hence maintains performance but it provisions for peak demand and therefore does not lead to high energy savings. B-dyn accounts for dynamic demands and significantly improves energy savings. B-fine takes advantage of fine-grained consolidation and increases savings even further, but since it only considers initial demands, it degrades performance. BLOC combines all techniques and leads to the best operating point, with 3.18x savings on average and performance maintained at 98.2% of that before consolidation. The difference between B-dyn and BLOC comes from the finer granularity at which data is placed. Smaller pieces of data offer more choices for placement, resulting in higher savings. Finer granularity improves server utilization but increases performance overheads due to increased inter-server communication, while coarser granularity achieves lower savings, as the ones shown in B-dyn.

Apart from average throughput and latency, worst-case performance is also important. Figures B.6b and B.6d show the throughput and latency CDFs for the most important schemes; we omit B-dyn and B-fine for clarity. Latency increase for BLOC is confined in a very narrow region around 0, and throughput degradation is minimal for most of the experiment's duration.

Node active times: Figure B.7 shows the cumulative distribution (CDF) of active time of the storage nodes in the cluster. Average active time drops by 61% with BLOC.

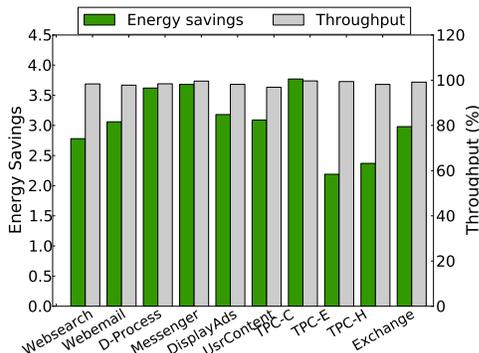


Figure B.8: Per application throughput and energy savings.

B.7.2 BLOC Behavior

Server utilization: Figure B.9 shows the operation of BLOC over a 24h period. Bandwidth and capacity utilization increases during consolidation for active nodes, but fewer nodes are utilized. Figure B.9a shows the bandwidth utilization for a representative node. Periods that had low utilization without consolidation (2am-8am & 8pm-2am) experience significant increase in utilization with consolidation. Also, since BLOC works adaptively, diurnal effects are less evident after consolidation. Figures B.9b-B.9e show how the load shifts to fewer nodes during consolidation. Average utilization increases by 45.7% in the active servers. BLOC tries to minimize node active times so most of the consolidated nodes are the same throughout the 24h period of the experiment. The wear-out and reliability of the nodes is often related to how many times they are power-cycled, and whether active nodes operate in a localized thermal hotspot. Appropriate methods can be used to ensure that active nodes are recycled in the cluster.

Application performance and efficiency: We previously saw the average performance and efficiency across all applications. However, given the different I/O access patterns of each workload, energy savings vary. Figure B.8 shows the throughput and energy savings for each of the ten applications, over a 24-hour period. In all cases, performance degradation is less than 3% since BLOC is careful in managing storage resource allocation. However, while all applications operate more efficiently, energy savings vary from 2.2x to 3.98x. This variance is expected since applications that

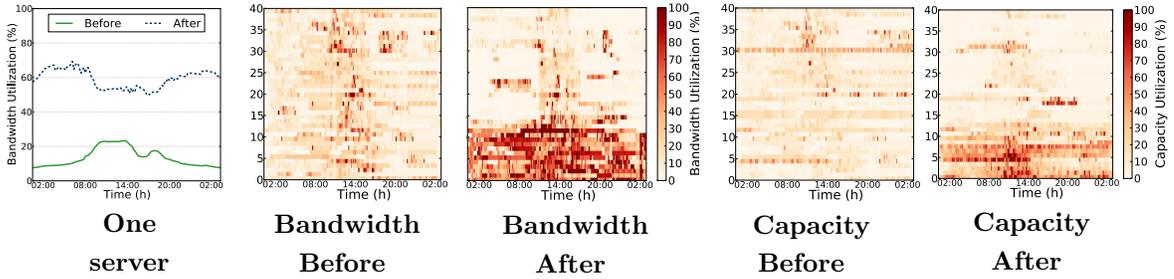


Figure B.9: Disk bandwidth (Figures B.9a-c) and capacity (Figures B.9d-e) over time, before and after consolidation. Utilization increases in fewer nodes after consolidation, while many nodes can be powered-down, improving resource efficiency.

stress the disk bandwidth more are less suitable for consolidation and achieve lower gains.

Data migration overhead: Figure B.10 shows the bandwidth used and amount of data migrated for the entire cluster over the 24-hour experiment. The upper graph shows the total bandwidth used, including the bandwidth for migration and bandwidth used to serve user demand. The difference between the two is small, indicating that the bandwidth used for data migration is negligible. The lower graph shows the temporal behavior of the data migration bandwidth. More migrations happens at the initial step of the experiment, when consolidation is first applied, and later in the morning when user demand rises and applications are deconsolidated. High migration is again seen in the evening hours as applications are consolidated back to fewer servers. The overall migration bandwidth is well below 5% except for the initial placement.

Parameter Sensitivity: One of the design parameters in BLOC is the frequency at which consolidation is updated. For our experiments we make consolidation decisions at a 30-minute granularity. This is based on the dynamics of the studied applications and the timing overheads of consolidation (time required for migration). Figure B.11a shows the throughput and energy savings when consolidation is reevaluated at intervals that range from 10 minutes to 2 hours. The optimal trade-off between performance and efficiency occurs for 30-minute intervals. Both metrics degrade for small intervals, due to the high overheads for data migration, and for large

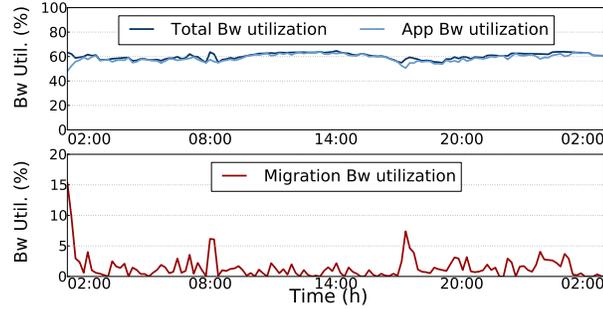


Figure B.10: Cluster-wide data migration over a 24h period.

intervals due to the overly coarse-grain estimation of resource requirements.

Another design parameter is the strictness of performance requirements. The maximum bandwidth and capacity limits, MB_i and MC_i , can be set higher for applications that are willing to accept lower performance. For instance, in Figure B.4a, rather than determining the bandwidth limit for a latency cut-off of 5ms (used as the performance constraint in earlier experiments), one could determine the bandwidth limit for a higher latency. Figure B.11b shows the energy savings for different performance constraints (percentages of original performance), e.g., 90% on the x-axis denotes that performance is 90% of the original, or degraded by 10%.

B.8 Discussion

Designing BLOC highlighted several issues that are interesting to explore in future work:

Bin-packing: The bin-packing problem used in BLOC is different from the multi-dimensional bin-packing problems considered earlier [210, 249] because in our setting, the size of the bin depends on the objects packed in it. Each time we place an application on a node, in addition to the reduction of available resources taken up by that application, the maximum limit up to which the node can be filled may decrease due to the new application’s requirements. The theoretical implications of this difference on the bin-packing problem and near-optimal heuristics are an interesting open

problem.

Replication and erasure codes: We described the consolidation algorithm assuming a replication factor of one. However, applications may use replication to meet bandwidth requirements, and may use replication or erasure codes for reliability. If replication is used for bandwidth requirements, previous methods [14, 258] can tune the number of online replicas to match demand, within reliability constraints. However, for reliability, the multiple replicas or erasure code blocks of any data item belonging to an application should not be placed on a single storage node. This constraint needs to be added to the consolidation algorithm in BLOC. One way to enforce it is that when a sorted list of the application objects is prepared for bin packing, the replicas or erasure code blocks should be maintained as separate lists. That is, if an application has n replicas or code blocks for each data item, then n lists are used. When packing objects to a node, objects are removed from only one list. Whenever a new node is started, the objects are selected from the list with the biggest head object. This way, reliability requirements are maintained while reducing resource costs.

Other storage systems: BLOC can be used with other storage systems, other than DFS. It only requires the address ranges, or data units located on each server. These can be physical blocks or logical blocks. For example, in the case of an RDBMS, data units would be tables of the relational database. Similarly, in the case of a distributed key-value store, such as Bigtable [51] the address ranges are shards or tablets. Finally, in the case of a virtualized system [130, 2, 273], since only entire VM instances can be migrated, a single VM would be the minimum consolidated unit.

Minimizing migrations: We greedily choose the applications with the smallest data footprints to be moved in the dynamic consolidation stage (Algorithm 2). This may not minimize the overall amount of data moved over multiple time steps. A longer term optimization that jointly considers placements over longer periods could be developed to reduce the number of migrations needed. Additionally, applications with small changes in demand could preferably be placed together so that those nodes do not experience frequent migration.

I/O pattern-aware consolidation: The maximum bandwidth for each node is

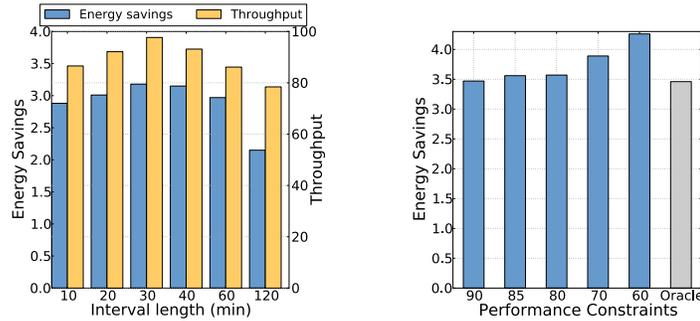


Figure B.11: Sensitivity to the length of time intervals at which consolidation decisions are made (Figure B.11a) and the strictness of performance constraints (Figure B.11b).

conservatively selected to the smallest value that preserves QoS for every application in that node. This maximum limit can be tightened to yield denser consolidation for applications where, exceeding this limit does not incur performance degradation (e.g., an application with random I/Os can be consolidated with a highly-sequential application without performance losses, even if the original bandwidth limit is exceeded).

Disk access reordering: Additional techniques, such as I/O access reordering [225] can be applied in BLOC to further improve the performance of consolidated applications. Reordering schemes can take advantage of the access locality recorded in the analytical model to determine the extent for which reordering is beneficial to performance.

B.9 Related Work

Consolidation is a well-known technique to reduce cost and energy consumption in DCs [14, 54, 126, 105, 125, 132, 140, 141, 163, 244]. We summarize related work below.

Capacity-based storage consolidation: Most prior work on storage consolidation is exclusively based on storage capacity requirements [140, 194, 195, 196, 197]. While they can reduce the number of storage nodes used, they ignore bandwidth requirements and hence result in performance degradation. Also, these techniques

perform consolidation at coarse granularity (entire application storage). BLOC accounts for bandwidth and partitions an application's data to perform consolidation at a finer granularity, which significantly improves efficiency.

Energy-proportional storage: Storage energy-proportionality can be improved using systems such as Rabbit [14], Sierra [258] and Pesto [131] that adapt to changes in load by tuning the replication factor. GreenHDFS [154] and Popular Data Concentration [215] employ techniques to identify and collect cold (unpopular) data on common servers and power those down. Data layout is optimized in [171] such that excess replicas can be powered down. BLOC addresses a complementary problem, since it reduces energy usage even when there are no excess replicas or cold data that can be taken offline, or when erasure codes are used instead of replication, as is the case with GFS [113]. Rather, BLOC allocates sufficient bandwidth to applications to serve current demand, and powers down excess resources. Section B.8 discusses how existing techniques such as [14, 258, 214] can be integrated in BLOC to manage replicas and erasure-coded data.

Storage virtualization: Storage virtualization mechanisms [130, 145, 181, 243] ensure that two or more workloads consolidated on the same storage node can each achieve their allocated bandwidths. However, these methods do not actually perform the consolidation or adapt to demand dynamics. Rather they can be used in BLOC after the consolidation has been performed to provide better performance isolation.

Consolidation policies: Extensive work has gone into bin-packing algorithms for server consolidation [10, 132, 168, 210, 244, 246, 249, 292]. Ajiro et al. [10] use an M/M/1 queueing model based on average resource utilization to determine the per-server consolidation density. Gupta et al. [132] propose consolidation in a virtualized environment by considering all resources as equivalent and static. Lee et al. [168] use vector bin-packing techniques for VM consolidation. Most of these schemes consider the processor, memory, and storage capacity requirements of consolidated applications, which are all largely additive. Storage bandwidth, is not additive, and is not considered in these systems. Also, consolidation is performed statically and does not consider migration overheads when the data layout changes over time. We dynamically adapt consolidation, while accounting for migration overheads.

Storage system design: Finally, there is work on storage design exploration such that the system configuration meets the requirements of a given application [17, 18]. These techniques also use bin-packing algorithms which, given an analysis of workload features, determine the appropriate storage system configuration. This work is orthogonal to BLOC. First, design decisions are driven by a performance model, which is determined for the storage load as a whole, as opposed to on a per-application basis. This is not accurate when multiple, diverse applications (e.g., Websearch and MapReduce) run on the same storage node. Second, this work assumes that the per-node storage is configurable, which is not the case for a large-scale homogeneous DC. Finally, it configures the storage system, e.g., RAID structure, to accommodate a given load, rather than decide which applications can be consolidated on a node of a given configuration. BLOC does not rely on performance models which might be proven inaccurate for complex multi-tier workloads, but rather determines consolidation based on available storage resources, such that there is no contention between applications. This enables higher resource efficiency (utilization), and stricter performance guarantees. However, assuming freedom in reconfiguring the storage system, BLOC can leverage some of these techniques to further boost resource efficiency.

B.10 Conclusions

We presented BLOC, an interference-aware storage consolidation system that improves DC resource efficiency and energy proportionality. BLOC leverages analytical models to determine the expected interference in the storage subsystem and the upper utilization limits that do not degrade performance for any consolidated application. It accounts for temporal variations in storage load, predicts future application activity, and reconfigures application placement to meet new demands. Experiments with real-world DC applications show over 3x reduction in energy consumption with minimal performance penalty. Compared to prior capacity-based methods, BLOC achieves high energy savings while maintaining performance constraints. We believe that the improved resource efficiency provided by BLOC can significantly improve the hosting costs of applications in large-scale datacenters.

Appendix C

ECHO: Network Modeling of Datacenter Workloads

C.1 Introduction

As the world's computation continues to migrate into massive datacenter (DC) infrastructures, developing highly efficient systems for these computing platforms has become increasingly critical. DC architectures are still in their relative infancy and when optimizing for performance, efficiency, or cost of ownership (TCO), architects must take into account the unique characteristics that dominate the behavior of large-scale applications. Understanding this behavior requires detailed workload characterization and is crucial not only from the systems but from the data analytics perspective as well.

DC applications are radically different from conventional workloads in several ways; first, privacy concerns make their source code, user behavior patterns and datasets rarely publicly available. This seriously hinders accurate and convincing studies. Second, DC applications experience activity patterns that cannot be reproduced or approximated by traditional benchmarks in standardized experimental environments because they only emerge from user behavior in the large scale, such as localized hotspots. Third, the cost of deploying experimental system configurations in a production environment is prohibitive both from the time and the cost

perspective. This increases the importance of concise, accurate and scalable models that representatively capture the behavior of large-scale workloads and can be used to create realistic access patterns.

The network component of DC applications reflects a large fraction of user patterns both in time and space. It is often responsible for Quality of Service (QoS) guarantees violations and accounts for a significant portion of the infrastructure's TCO. Despite the obvious merit in developing representative analytical models that capture DC network traffic, unfortunately previous work lacks the ability to reflect the complex spatial and temporal patterns that emerge in the large-scale. For a workload model to be useful in the context of large-scale DCs it needs to have three main properties: (a) *accuracy*, so that the information in the model closely resembles the actual behavior of the application, (b) *modularity*, so that it can: (i) adjust the granularity of information to the needs of an application and (ii) be reconfigurable and compatible with other components, and (c) *scalability*, in order to capture large-scale effects in a lightweight manner. Existing solutions either fail to capture the spatial patterns in network traffic or introduce significant computational overheads and are unable to scale past a few servers.

In this chapter, we present ECHO, a scalable and accurate modeling scheme that captures the spatial and temporal behavior of network traffic in large-scale DC applications. ECHO is derived from *validated* analytical models that enable it to concisely represent the network patterns of workloads, while provably guaranteeing low and upper-bounded errors. As part of ECHO, we present two models; first we examine a simple, distribution fitting model that captures and generates per-server network traffic by recognizing known distributions in network activity fluctuation. To capture the burstiness and *spatial patterns* of network load, e.g., server-to-server traffic, we propose a Markov Chain model that is topology-independent and locality of communication-aware and captures individual server interactions. Additionally, we make ECHO hierarchical, adjusting the level of detail in the model to the requirements of each application. Starting from groups of racks and increasing the level of detail down to individual servers, ECHO captures and recreates the spatial patterns of network activity in DCs with tens of thousands of servers. We perform a detailed

validation study and show that the deviations between original and generated traffic are marginal for the network activity of two large-scale systems. We also verify that ECHO captures all the critical features of DC applications, such as spikes in network activity and inter-, intra-rack communication. Additionally, we perform a detailed characterization of the temporal and spatial patterns of the network activity of DC applications in three DC deployments over a period of five months.

C.2 Related Work

Network Workload Characterization: The network is one of the most widely characterized aspects of a workload since it reflects user patterns that emerge in the application. This characterization becomes more interesting for DC workloads running on tens to hundreds of thousands of servers. Although there is extensive prior work on network characterization for traditional applications [27, 46, 92]; in this chapter we focus on related work in the context of large-scale DC systems.

Feitelson [104, 103] presents a detailed characterization of network requests based on their stationarity, self-similarity, burstiness, and heavy tails; features that dominate DC workloads. Yu et al. [287] use a detailed profiling of the TCP/IP stack to troubleshoot network performance problems of multi-tier DC applications. They design a generic, app-independent profiler that monitors TPC at the socket-level and identifies performance bottlenecks within the same and across different connections. Ersoz et al. [95] also characterize the network traffic of a multi-tier DC. They observe that inter-arrival times and message sizes follow log-normal distributions, while service times fall within the Pareto distribution and show heavy tails at heavy loads. Benson et al. [37] analyze the features of network traffic in several cloud DCs classified per application type, in terms of temporal patterns, network and link utilization, congestion and packet drops and in [38] propose a fine-grain scheme for traffic engineering in these systems.

Atikoglu et al. [23] perform a workload analysis of Facebook’s key-value store, *memcached*. They observe that GET requests by far dominate over SETs, while

spatial locality widely varies across memcached servers. They also record user patterns that experience diurnal behavior and propose statistical modeling to extract the distribution of request inter-arrival rate. Also in the area of large-scale workload analysis, Shafiq et al. [236] study the machine-to-machine (M2M) traffic in cellular networks which has similarities with the traffic of certain user-interactive DC applications. They characterize the temporal dynamics (e.g., diurnal behavior, burstiness), hot spots, application usage and data upload/download of a dataset from a large network service provider. Many of their findings are consistent with the behavior we present in Section C.4 for latency-critical DC applications. Using network characterization in a different scope, Gill et al. [116] measure DC network load to evaluate the system's reliability and characterize the components most prone to fail.

Network Workload Modeling: Similar to characterization, network modeling has attracted significant interest due to the user patterns of large-scale DC applications. Feitelson [104] apart from characterizing network loads, presents an overview of methods to model network request distributions. He suggests distribution fitting through the Kolmogorov-Smirnov test, to identify known distributions in network traffic fluctuation. Furthermore, he presents preliminary considerations on the correlation between task size, arrival rate and execution time when combining network and CPU modeling. Although his work presents strong arguments for the value of modeling temporal variations of network traffic, it does not present any validation of the proposed techniques against actual applications. In Section C.3 we evaluate a similar distribution fitting-based model, but additionally validate it against real DC workloads. Building from this paper, Li [173] characterizes network and CPU-intensive applications running on large-scale grids. He analyzes features like job arrival rate, size, pseudo-periodicity and correlation of these features with execution time. He, then, proposes a two-phase approach to model these workload attributes. The first step consists of *Model-Based Clustering* which performs distribution fitting. The second step generates autocorrelations to create synthetic workloads that resemble the original load. Although this work provides some insight on the performance impact of request distributions in grids, it is computationally intensive and does not scale beyond a few nodes, making it inapplicable in large-scale DCs.

Barford et al. [27] study the characteristics of Web servers and propose a workload generator that recreates patterns with temporal fluctuation and request sizes similar to the original application. Joo et al. [151] propose network traffic modeling to identify and resolve performance bottlenecks in a small machine cluster. They compare two different models, an *infinite source-based* model that is user-invariant, i.e., all users send the same amount of data, and a *SURGE-based* model, where traffic varies per user. They observe that ignoring user variation and information about network topology causes significant inaccuracies in the generated workloads. Sengupta et al. [234] characterize the request arrival rates of a series of OLTP workloads and propose an analytical distribution fitting and self-similarity recognition model. They conclude that accurate modeling of network traffic can facilitate decision-making for a series of performance/energy-related optimizations, although their study is only limited in transaction-based applications running on a small cluster. Similarly, Danzig et al. [67] propose an empirical workload model based on statistical analysis of wide-area TCP/IP traffic. They use both inter-arrival times to model traffic that follows known distributions and amount of data transferred to model burstiness. Finally, Tang et al. [256] propose a framework that captures non-stationarity, burstiness and request duration to model long-time network behavior. Based on this model, they develop MediSyn, a publicly available streaming media workload generator.

Overall, the common base in previous work is a focus on capturing temporal variations in network load using analytical models such as distribution fitting. Although this can provide insights on load variations and identify user patterns over time, it includes no notion of spatial patterns, which are a crucial part of DC applications. Hot spots due to load imbalance, inter-application traffic and server-to server communications are all features that impact critical design decisions in large-scale DCs and necessary for a modeling scheme to capture.

C.3 Single Server Temporal Model

We first focus on the requirements for simplicity and conciseness of the scheme, with a simple model that captures per-server network activity over time. We show which

features of network load this model captures and why on its own it is not sufficient to model the behavior of DC applications.

We adopt a distribution fitting model which has been previously shown to be useful to capture the behavior of conventional network workloads [173, 234]. In this work we are validating its accuracy in the context of a real large-scale DC application. The model takes as input a network bandwidth trace from a single server and identifies known distributions (e.g., Gaussian, Poisson, Zipf, etc.) in the activity pattern. The output is a mathematical expression that is the superposition of identified distributions similar to Expression C.1.

$$\begin{aligned}
 BW_o &= \sum_{i=0}^N (\text{Distribution Expressions}) = \\
 &\sum_{i=0}^{N_1} f(x; \mu, \sigma^2) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu_i}{\sigma_i}\right)^2} \Big|_{t_{i\text{start}}}^{t_{i\text{stop}}} + \\
 \sum_{i=0}^{N_2} f(k; \lambda) &= \frac{\lambda^k e^{-\lambda}}{k!} \Big|_{t_{i\text{start}}}^{t_{i\text{stop}}} + \sum_{i=0}^{N_3} f_{\text{other}} \Big|_{t_{i\text{start}}}^{t_{i\text{stop}}} + \dots
 \end{aligned} \tag{C.1}$$

where N is the number of identified distributions and N_i the number of individual distributions of each type.

We validate the accuracy of the model by comparing the network traffic generated based on the model against original traffic patterns. Figure C.1 shows this comparison for a Webmail workload running on a production-class server. The deviation between original and generated load is less than 4.9% on average ensuring that the model accurately captures temporal variations in network load. In this case the model identifies three Gaussian, one exponential and one constant distribution in the network activity. We have performed additional validation experiments with workloads that experience diverse activity patterns and verified the consistency of the results.

Although the distribution fitting model is a simple and comprehensive way to capture the load of individual servers it is agnostic of the source and destination of traffic, therefore it cannot represent spatial effects, such as server-to-server communication. Additionally DC workloads often experience short bursty periods in their

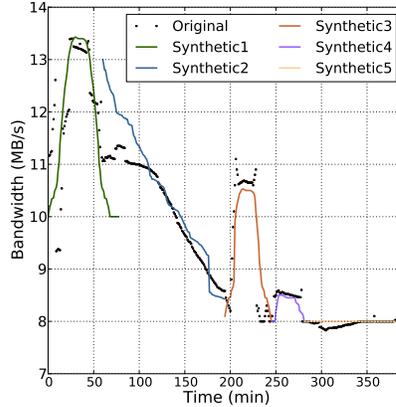


Figure C.1: Distribution fitting model validation.

network activity [104], which cannot be accurately approximated by known distributions. The next Section describes the most critical features of network workloads in the large-scale that the model should accurately capture. To recreate them we adopt a different approach. The hierarchical Markov chain model described in Section C.5 is a scalable, yet accurate solution that captures both temporal and spatial effects without becoming intractable in complexity.

C.4 Temporal and Spatial Network Traffic Characterization

Previous work has established some trends for DC workloads, such as diurnal behavior and variation in activity between different time intervals, e.g., weekdays over weekends [23, 37, 95]. Here we validate these findings and additionally perform a detailed study on the spatial locality of network traffic in DC applications, as well as its fluctuation over time. We examine the network load of two production DC deployments; a system with several tens of thousands of servers from Microsoft running Websearch and two smaller systems with several hundred and a few thousand servers respectively running a single application. Specifically, the first system runs a combiner for query results, part of Websearch (Combine), while the second system

extracts a snippet of information from backend Websearch servers and displays it to the user, in the top of the search results (Render).

Fluctuation of network activity over time: We show network traffic over time in the form of heatmaps. Each tick on the x-axis represents an interval of 5 minutes and each point on the y-axis, a server ordered by server id. Consecutive servers belong to the same rack in groups of 48 servers per rack. The colorbar on the right shows the range of network traffic bandwidth observed. Darker color in the heatmap represents higher network traffic. There are a few servers in each heatmap that exceed this range, however they represent a very small percentage of the total deployment, typically 0.1%-0.5%. Figure C.2(a) shows the fluctuation of activity per server in the large DC running Websearch over the period of one month (December 2011). As shown in the graph, the network load varies across groups of servers, with specific nodes experiencing high traffic, although mostly the system remains well load-balanced. Overall, the network traffic is very low; very few nodes exceed 0.0064MB/s, which is in agreement with the extensive overprovisioning present in large-scale deployments of latency-critical applications such as Websearch. Additionally, the traffic over time remains mostly self-similar, consistent with well-known DC application characteristics [23]. The diurnal patterns in network activity, i.e., difference in load between day and night, become more clear in Figure C.4(a) and (c) which show per-server traffic for Combine and Render. Every dark vertical band representing the hours of the day is followed by a lighter interval (e.g., in the x-axis: vertical band at range 970-1030) representing the low activity period of the night. There are 31 dark and 31 light bands in total in the graph representing each 24h period in the month. Compared to Websearch, both Combine and Render have significantly higher network traffic, as a result of the lookups necessary to extract and aggregate search results. Between Combine and Render, the former has slightly higher network activity, since larger chunks of data are transferred between servers. To better visualize the diurnal patterns in network activity and their change, in Figure C.4 we plot network load over time, averaged across all servers. Figure C.4(a) shows the fluctuation in load for Websearch across five consecutive months (November 2011-March 2012). Overall, the patterns are similar, although their magnitude varies across different months, with

significantly higher loads in December and January. Figure C.4(b) shows the per-week breakdown of network load for December. Again, diurnal patterns are present, with load being 10-15% higher in the last two weeks of the month, and 15-18% higher in the days of the weekend, compared to weekdays. Finally, Figure C.4(c) shows the per-day breakdown for the last week of the month. The load experiences two peaks, one from 12pm to 6pm and another from 8am to 12pm while progressively decreasing in the hours of the night.

Although the time aspect of DC applications has been extensively studied, the same is not the case for the locality of communication in network applications. This study has three types of contributions; first, it can verify that the application takes advantage of *data locality* by confining most requests to servers of the same or neighboring racks to reduce latencies, second, in the presence of multiple applications, it can provide insight to the *scheduler* on how to assign machines to jobs to confine inter-application traffic to neighboring nodes and third, it can verify that *load balancing* prohibits the formation of extensive hot spots. In this work, we examine both the average locality of network traffic and its fluctuation over time to provide insights on these issues.

Spatial locality of network activity: Figure C.2(b), Figure C.3(b) and (d) show the server-to-server network traffic for Websearch, Combine and Render. The x and y axes represent servers and the graphs are symmetric over the principal diagonal since, for example, traffic between servers 2 and 3 is the same as traffic between servers 3 and 2. This includes both sent and received traffic for each pair of servers. More prominently in Websearch, but for the other applications as well, the majority of traffic is confined within servers of the same or consecutive racks, seen by the darker colors along the diagonal. For all three applications, especially Combine and Render, there is a small number of servers that talk to most of the machines in the system. These are likely servers running the cluster scheduler, aggregators (in multi-tier Websearch) or monitoring systems.

In the smaller clusters of Combine and Render, we see less network locality, as several servers, especially in the first few racks, exchange requests with machines outside their rack. This is consistent with the functionality of the two applications,

aggregating and caching or displaying results extracted from search queries. Overall, we observe that in Websearch, which is dominated by query latency, the job scheduler is data locality-aware and tries to minimize long request round trips to servers that are far from each other.

Finally, we examine the actual network traffic map for Websearch in Figure C.2(c), where each shaded region corresponds to a rack, with 48 servers within each region. All servers in a rack are connected to the same switch, while servers in different racks are connected to different switches. As seen in the figure, with the exception of few, most racks have similar levels of network load, verifying that the load balancer assigns work to machines preserving fairness, while the small number of racks which have slightly higher network traffic are the ones that talk to many servers in the DC.

Fluctuations in the locality of network activity: Apart from examining the average locality of network activity, we look at how these patterns change over time. Figure C.5 shows the server-to-server network activity for Websearch over a period of five consecutive months in 2011 and 2012. In most cases the patterns remain consistent across different months with a slight surge in traffic in December and January, which is in agreement with our findings for the activity fluctuation over time for the same period (Figure C.4). When comparing this to the fluctuation in locality observed in Combine in the same period (Figure C.6) there are significant differences. Unlike Websearch, Combine experiences significant differences in its locality over different months, probably the result of changes in the application's structure and functionality. For example, while in November and December there is a significant number of servers polling a large fraction of the system, in the following months the requests become progressively more localized in servers of the same or neighboring racks. March also experiences an interesting access pattern with high localized network activity in specific servers (range 180-250).

Although at the month granularity Websearch seems invariant in its locality patterns, when moving to a finer granularity we see that it also experiences fluctuations in its network activity locality. Figure C.7 and C.8 show the server-to-server traffic for two weeks of December and three days of the second week respectively. At the

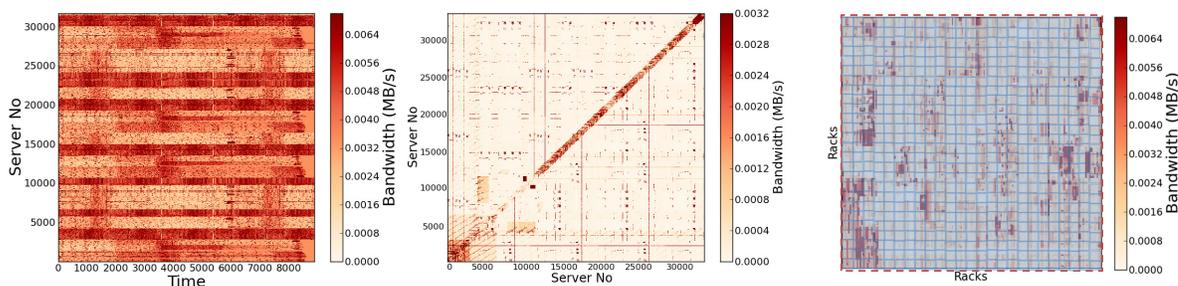


Figure C.2: (a) Per-server network traffic over a 1 month period, (b) server-to-server traffic and (c) rack-level network traffic map across the entire DC for Websearch.

day granularity, we observe that there are significant differences in the locality of accesses, with different subsets of the DC being more active than others, however these fluctuations get hashed out at large time intervals.

Overall, we observe that the network activity of large-scale applications changes both in time and space, and for a model to provide useful insight in the behavior of the application, it should capture the workload accurately across both these axes.

C.5 System-Wide Spatial Model

C.5.1 Overview

From the previous Section there are several network activity features that are critical for a workload model to accurately capture; (a) the *average activity* (per-server and system-wide), (b) the *fluctuation of activity over time*, both bursty and with defined intervals, (c) the *locality of activity*, i.e., spatial patterns across the system and (d) the *interactions* between specific servers or racks. This is a wide and often conflicting set of requirements to be met; for example a model that captures spatial distribution of load might not capture temporal variations accurately.

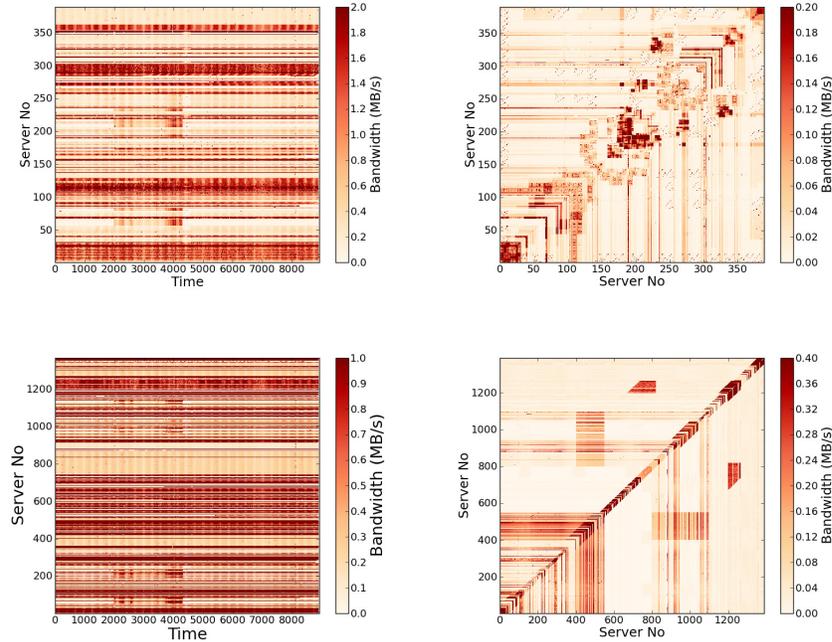


Figure C.3: Network traffic over time and server-to-server traffic for Combine and Render.

C.5.2 Design

To address these requirements we propose ECHO, a scalable modeling scheme that captures both temporal and spatial network patterns. ECHO leverages robust analytical models which enable it to provide strict error bounds on the accuracy of the representation. Specifically, it is driven by a set of Markov chains which are trained on real input traces from production DCs and provide a probabilistic framework to both characterize and recreate network activity patterns.

To tackle the scale of modern DCs, ECHO is hierarchical, starting from groups of racks, and modeling network activity down to racks and individual servers. This way the level of granularity is adjusted to the specific patterns of a given application. ECHO is topology-independent, since the aggregation of state in the model is not tied to a specific network organization, therefore it is portable across different topologies, provided no radical changes in the system’s organization. Figure C.9 shows the hierarchical probabilistic model for a system with four groups of racks. Each state

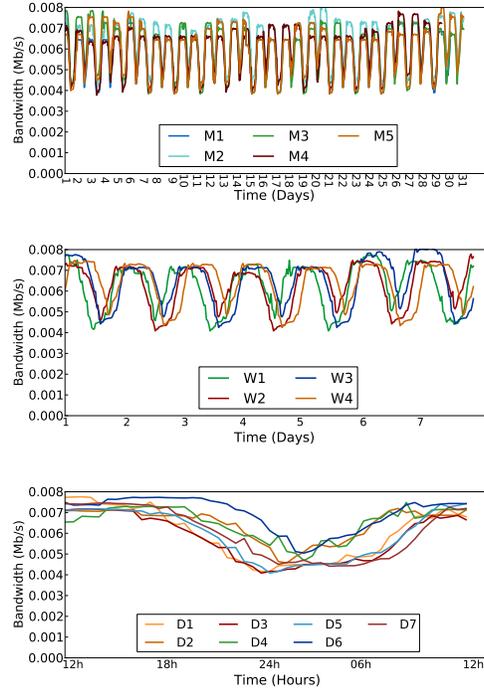


Figure C.4: Network activity averaged over all the servers in the system across (a) five months, (b) four weeks and (c) seven days.

represents different entities in each level. In the highest abstraction, a state is a group of racks, while at the second level it represents a rack, and at the lowest level it represents an individual server. The number of groups of racks or racks in each group is a configurable parameter of the model, while the number of servers per rack is a design parameter of the system. A transition between any two states represents the probability/portion of network traffic between two entities that has certain characteristics. These characteristics correspond to the size of network requests, the type and the inter-arrival times that separate them, or their burstiness. The Markov chain defines such probabilities as:

$$p_{ij} = Pr[X_j|X_i] = Pr[Server_j \leftarrow Server_i \mid MB/s, rd/wr, int. time] \quad (2)$$

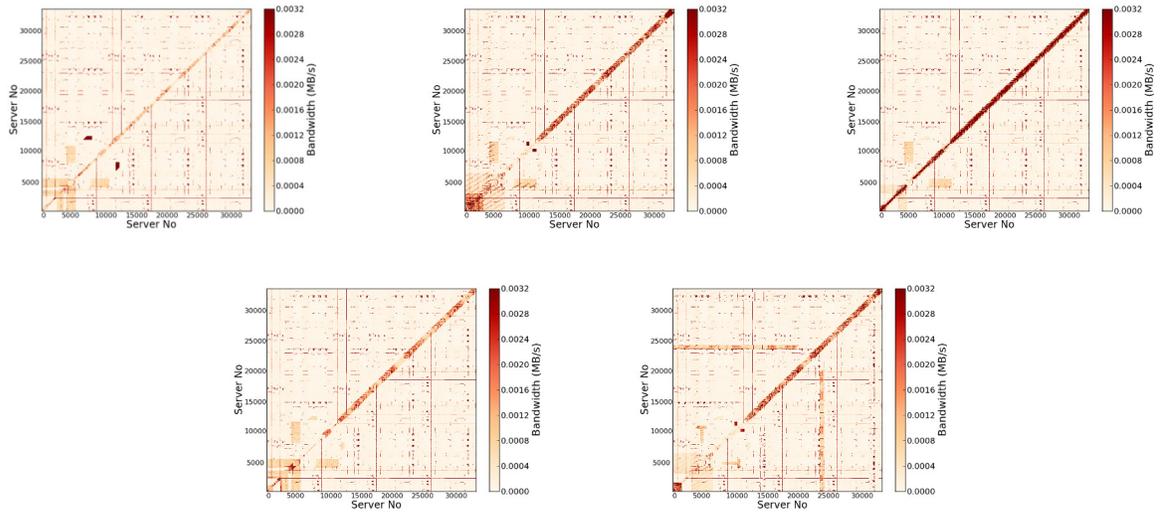


Figure C.5: Server-to-server traffic for Websearch over a period of five consecutive months in 2011 and 2012.

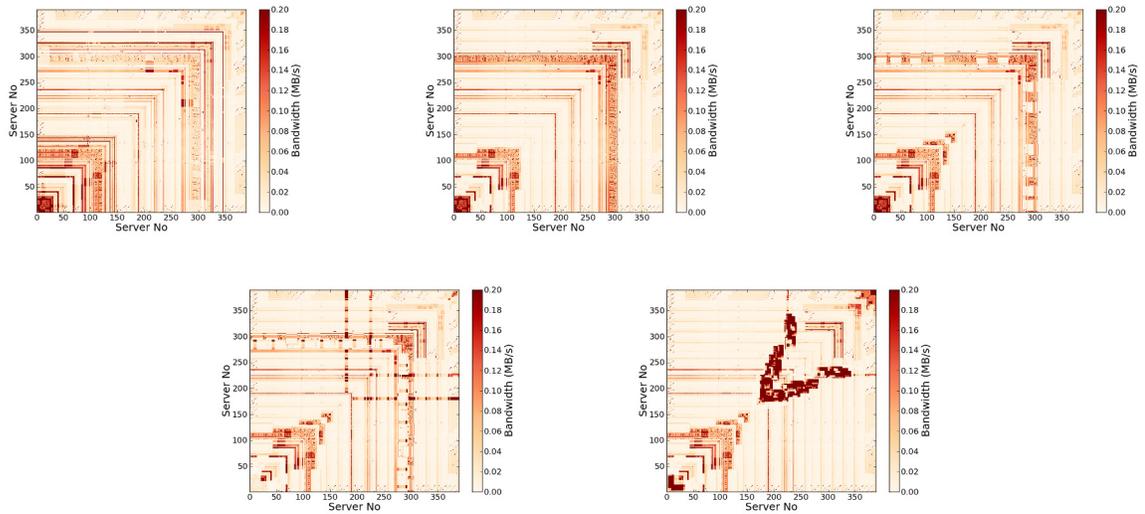


Figure C.6: Server-to-server traffic for Combine over a period of five consecutive months in 2011 and 2012.

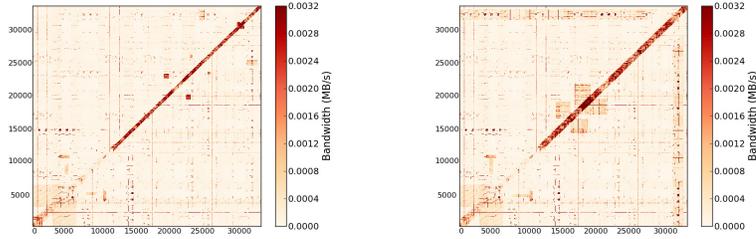


Figure C.7: Spatial locality of network activity across two different weeks in December 2011.

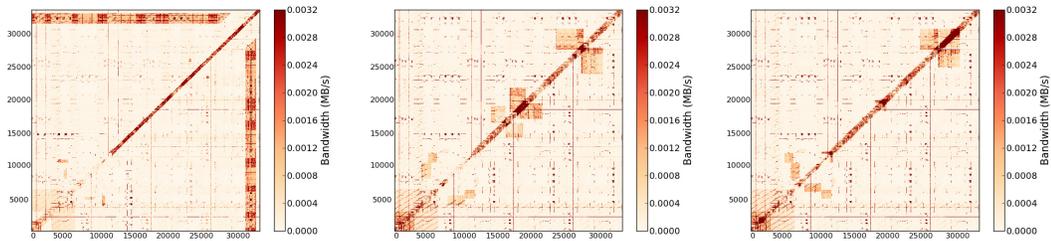


Figure C.8: Spatial locality of network activity across three different days in December 2011.

where there is a p_{ij} probability that $Server_i$ interacts with $Server_j$ over the network with specific bandwidth (MB/s), request type (rd/wr) and inter-arrival time. Although the use of both inter-arrival times and probabilities might seem redundant, it enables modeling both requests that follow well-defined distributions (in which case the two metrics are almost equivalent) and bursty behavior (when the inter-arrival rate is very high, but the probability of the traffic occurring low).

For the clarity of the representation some transitions are omitted from Figure C.9. Based on the workload characterization of Section C.4, we observe that the majority of network traffic, with a few exceptions, is confined within the servers' corresponding rack. This motivates the use of a hierarchical rather than a flat model where all transitions are explored. As seen in Figure C.9 transitions between large states remain coarse-grain when moving to lower levels of granularity, while transitions within large states are expanded. This improves the scalability of the scheme, by enforcing upper bounds for the transition count and prevents a complexity explosion when scaling up to thousands of servers, while conveying the same amount of information about

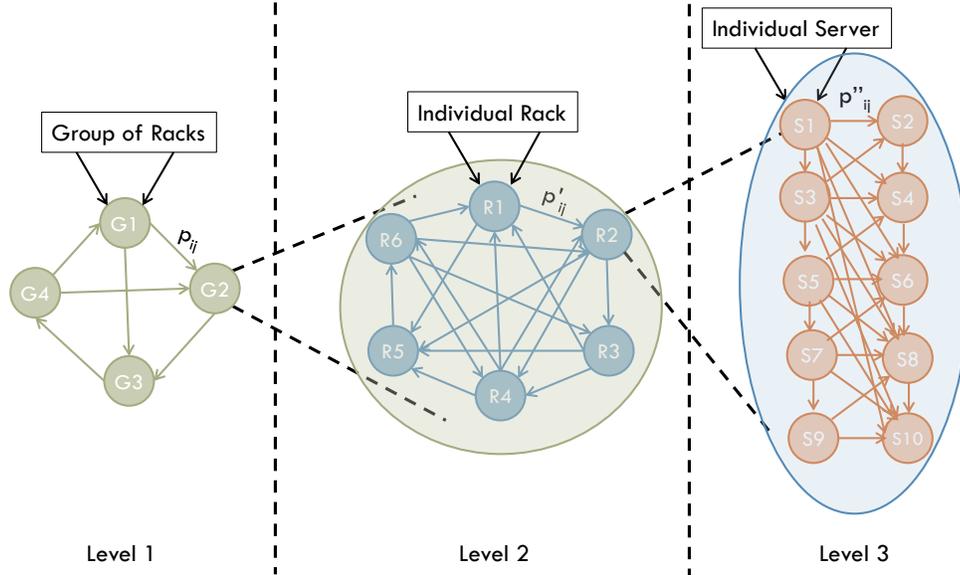


Figure C.9: Schematic of the hierarchical spatial Markov chain model.

the workload. Comparing a flat and hierarchical model for a system with 100 racks reveals a 90% reduction in transition count for the hierarchical model.

The model described so far captures both spatial and temporal patterns in a DC for a given time frame. However, network activity is highly volatile switching from low to high load frequently and experiencing short periods of high burstiness in the requests. To capture such evolving patterns we rely not on one, but on a set of Markov chains, each one corresponding to a different interval of the application's execution. The number of models is a configurable parameter, depends on the characteristics of the application and since individual models can be created in parallel, higher disaggregation in time not only does not increase, but decreases the overhead of the modeling process. For example, to create a single Markov chain a system-wide bandwidth trace is required. Parsing the entire trace, although only done once can introduce some overhead in the modeling process. However, partitioning this procedure to create multiple Markov chains for different time periods and processing them in parallel, both reduces the training overhead and improves the convergence between original and generated load. When generating a synthetic workload, the different models are activated in a Round-Robin fashion based on the time frame they correspond to, e.g., when there are separate models for each 12h period of a day,

50% of the time the generated workload complies with the first model and 50% of the time with the second model. Finally, although the model is non-deterministic, which means that subsequent runs may be different, we verify that each one of them maintains close resemblance to the original workload.

C.5.3 Validation

For our validation we focus on four main metrics, as discussed in Section C.4; (i) average network traffic, i.e., *how much traffic is generated on average irrespective of temporal and spatial patterns*, (ii) per-server activity fluctuation over time, i.e., *how much traffic*, (iii) spatial patterns of network activity, i.e., *to whom is the traffic sent* and finally a more fine-grain metric, (iv) individual server interactions, i.e., *how much traffic, to whom and when*. We show validation results for the hierarchical spatial model using as input real network activity traces from two production DC deployments with hundreds to tens of thousands of servers. Unless otherwise specified all results are using the 3-level hierarchical model. Additionally, experiments shown in this work are repeated over five runs to verify the consistency of the results.

Average activity validation: We evaluate the model against two types of systems. First, we validate the model in a small test cluster of a single rack with 24 servers. The workload is the network traffic of servers hosting the webmail of an academic institution. We examine four two-hour instances of this load; Load A is the load on the morning of a weekday, Load B the load on the afternoon of a weekday, Load C the load on a night of a weekday and Load D the load on a weekend. Figure C.10 shows the comparison between original and synthetic traffic generated based on the spatial model for Load A (the results are similar for the other loads) and the table in Figure C.1 the statistical metrics that quantify this deviation. In all cases the difference between original and synthetic workload is less than 7% validating the accuracy of the model.

Second, we validate the model against real DC workloads on two production systems; the DC running Websearch on several tens of thousands of servers and the smaller cluster running Combine. The temporal patterns in Section C.4 reveal that

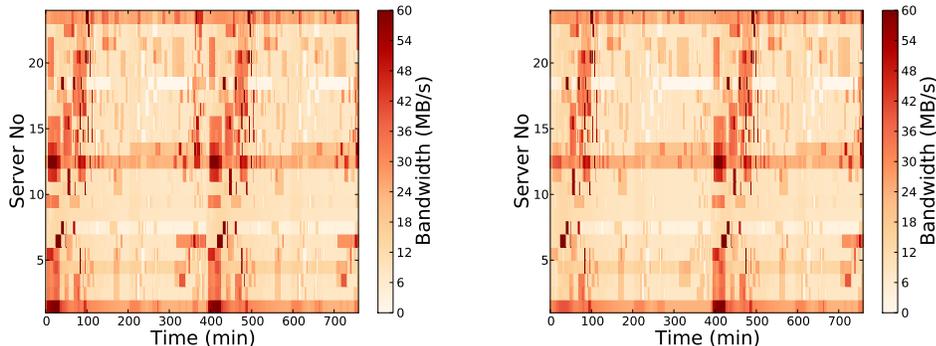


Figure C.10: Per-server bandwidth comparison between original and generated network traffic for a one-rack system running Webmail at load A.

load changes significantly both throughout a 24h period, and between different weekdays compared to the weekend. Based on these patterns we create a set of 10 models for each system; 4 models from six hours periods of two different weekdays ($2 \cdot 4$) and 2 models for 12h periods of a day of the weekend. This choice is configured to the features of the specific applications and can be different for other workloads. We first validate the average statistics of network activity between original and generated workloads. Table C.2 shows the deviation in average traffic both for fractions of servers and system-wide. The first 4 columns show the deviation for the 25, 50, 75 and 95 percentages of servers, sorted by increasing average network load and the last one shows the average deviation across the entire system. In all cases, the deviation is marginal, with the system-wide being lower than the ones for server subsets for both applications. We also observe that the lower the load of a server the higher the deviation between original and synthetic workload, while for higher loads the deviations are lower. Additionally the larger the modeled system the lower the deviation.

Activity fluctuation validation: We validate the accuracy of the model in capturing the fluctuation of network activity by generating synthetic workloads for subsets of the two production DCs, Websearch and Combine. There are two reasons for not validating the model against the entire DC; first, the unavailability of an equally large system to run the generated workload and more importantly the fact that although possible, system-wide generation of a workload obscures some of the value of the model, which is identifying interesting patterns or performance issues in network

Server Percentage	Average Deviation			
	Webmail			
	LoadA	LoadB	LoadC	LoadD
25%	3.8%	4.3%	2.3%	2.9%
50%	4.2%	5.6%	2.5%	3.4%
75%	4.5%	6.2%	2.7%	3.5%
95%	5.5%	6.3%	3.1%	4.2%
All Servers	5.8%	6.3%	3.0%	4.1%

Table C.1: Validation of average statistics between original and generated network load.

Application	Average Deviation for Percentages of Servers				
	25%	50%	75%	95%	System-Wide
Websearch	7.2%	5.9%	4.2%	3.9%	3.8%
Combine	8.5%	6.9%	5.3%	4.4%	4.4%

Table C.2: Validation of average statistics between original and generated network load for Websearch and Combine.

traffic and reproducing them independently, without full application deployment. The validation process works as follows:

- We select subsets of servers in the original workload that present interesting traffic patterns.
- We create models that are trained on network activity traces from each subsystem.
- We generate a synthetic workload for each subsystem based on the corresponding model.
- We compare the network traffic between original and synthetic workload.

Figure C.11(a) shows this validation for Websearch and Figure C.11(b) for Combine using as input the network activity over a one month period (December 2011). This activity for Websearch includes traffic from multiple components of the application, such as indexing, advertisements and answers to user queries, while for Combine it represents network traffic from a single application. We have selected four server subsets in each case, representing machines with both low and high activity, as well as static or varying traffic patterns over time. The figures visually demonstrate the

similarity between original and synthetic patterns, while the corresponding Table C.3 quantitatively confirms these findings. In all cases, the errors both for percentages of servers and across the entire sample are lower than 7% for Websearch and 8% for Combine, verifying the accuracy of the model in capturing time variations.

An important point when modeling system subsets is that traffic is not necessarily self-contained in them. For example, if a subsystem includes servers 1-10, it is not necessarily true that server 1 does not interact with server 11 and vice versa. Therefore to maintain the accuracy of per-server load when choosing parts of the system, we need to ensure that the external traffic is also represented in the model. We do this by introducing a single "cloud" node that generates and receives all traffic from and to servers in the subsystem. Figure C.14 shows how this external node is integrated in the model. Essentially the model has one additional state which is responsible for the network traffic outside the selected subsystem.

Validation of locality of network activity: A critical goal of the model's design is capturing spatial patterns in network activity. We verify this with a similar experiment to the one previously described. We select four subsets of servers from the server-to-server traffic maps for Websearch and Combine and generate synthetic workloads based on the corresponding models. Figure C.12 shows a visualization of the similarities between original and synthetic subsets and Table C.4 presents the quantitative metrics that confirm these similarities. For both systems the deviations are low, with higher load servers typically experiencing smaller inaccuracies. Overall the deviation is less than 10% for Websearch and 11% for Combine.

Finally, Figure C.16 shows the error CDFs for the temporal (Figure C.16a) and spatial (Figure C.16b) patterns of the generated workload against the original application. The errors reported are for the large-scale DC running Websearch. From the figure, the errors for spatial patterns are slightly higher than for temporal patterns, but in all cases the 50th percentile of servers have less than 4% error in temporal and less than 5.2% in spatial while the 90th percentile of servers have less than 7.3% in temporal and 9.8% in spatial. Only few outliers experience errors larger than 10%, validating the accuracy of the modeling scheme.

Application	Average Deviation (%) for Percentages of Servers in each Subset																			
	Subset S1				Subset S2				Subset S3				Subset S4							
	25	50	75	95	100	25	50	75	95	100	25	50	75	95	100	25	50	75	95	100
Websearch	1.2	2.4	5.7	6.8	6.6	3.9	3.8	4.7	5.3	5.2	3.2	3.6	3.7	4.1	4.0	7.2	6.7	6.8	7.1	7.0
Combine	2.3	3.5	3.6	4.4	4.2	4.2	3.8	3.6	3.7	3.8	5.1	4.8	4.9	4.6	4.5	7.6	7.9	8.1	8.0	8.0

Table C.3: Validation of generated network activity over time for server subsets in two DCs.

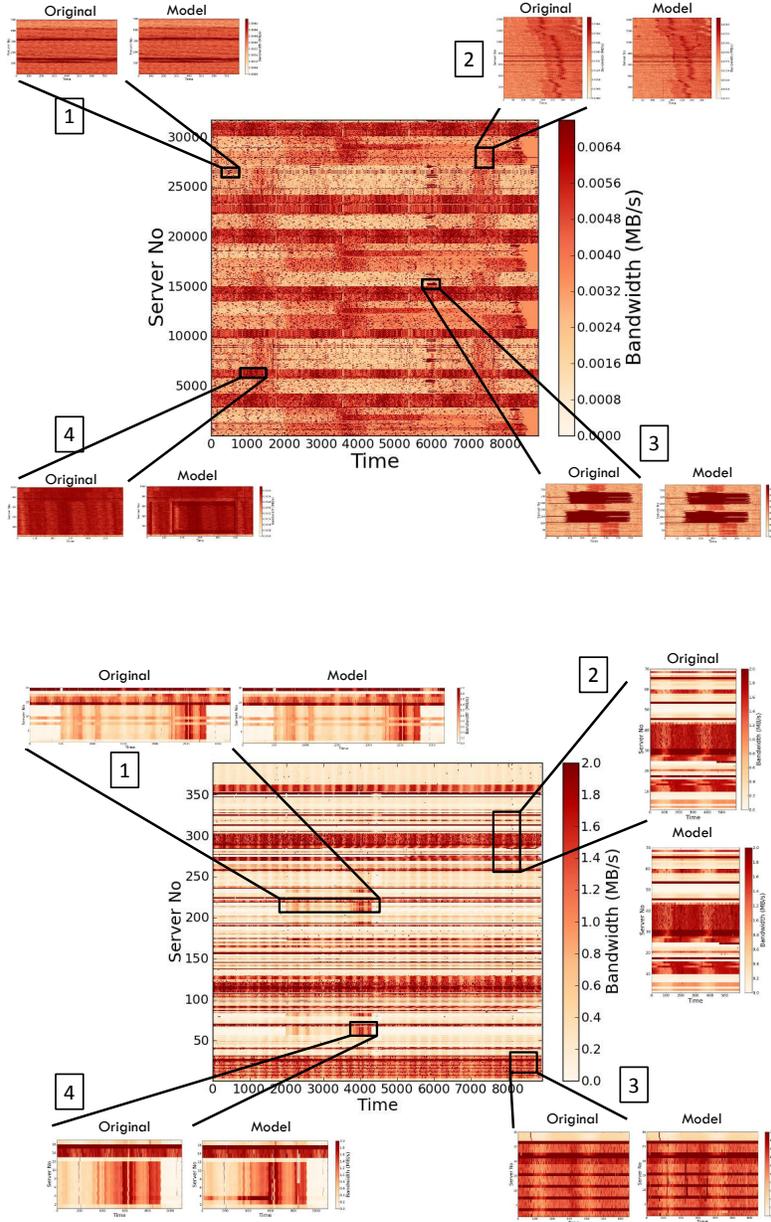


Figure C.11: Spatial model validation for server activity over time. Figure C.11a shows the validation of ECHO against subsets of the large-scale DC running Web-search and Figure C.11b a similar validation for Combine running on a smaller DC.

Application	Average Deviation (%) for Percentages of Servers in each Subset																			
	Subset S1				Subset S2				Subset S3				Subset S4							
	25	50	75	95	100	25	50	75	95	100	25	50	75	95	100	25	50	75	95	100
Websearch	9.5	5.2	3.4	3.2	2.7	6.2	5.4	4.9	4.6	4.6	4.6	4.2	4.1	4.3	4.3	4.4	10.8	9.9	7.6	8.2
Combine	6.5	4.9	4.3	2.4	2.4	3.4	3.5	4.7	4.6	4.5	5.8	6.7	7.3	6.3	6.4	11.2	10.8	8.2	5.6	5.4

Table C.4: Validation of generated network traffic map for server subsets in two DCs.

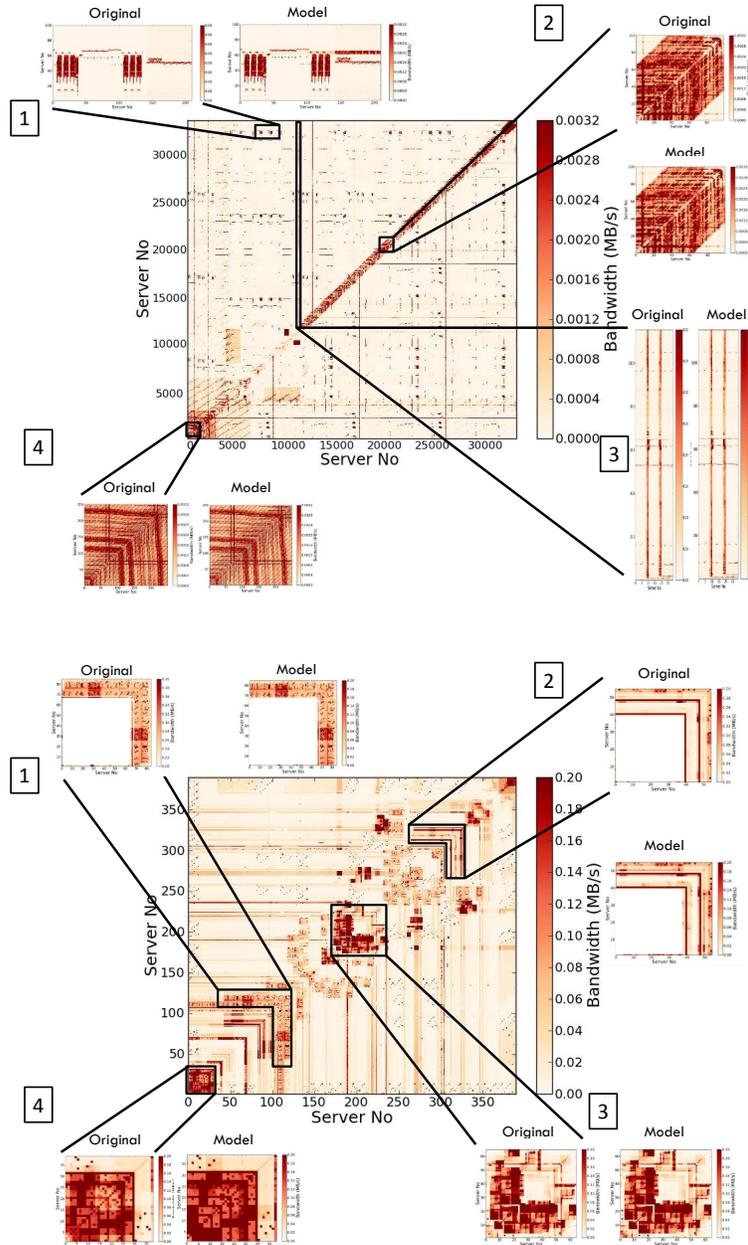


Figure C.12: Server-to-server network traffic map validation for server subsets of (a) Websearch and (b) Combine.

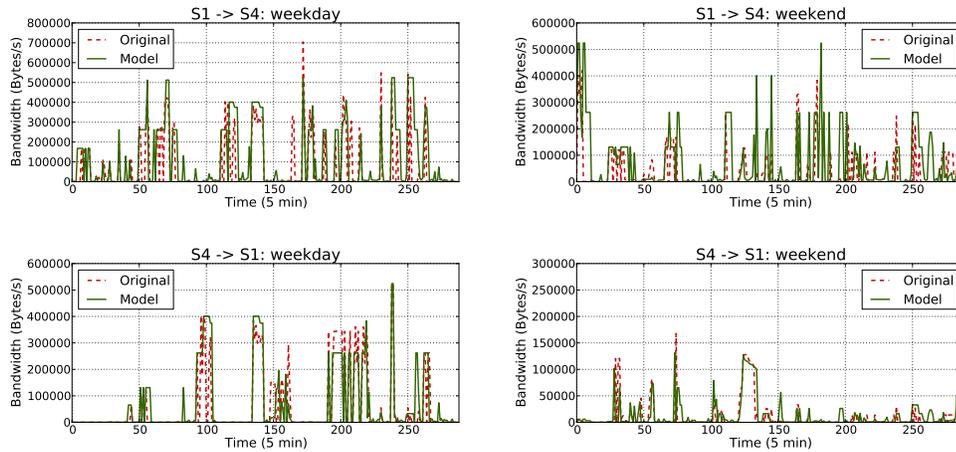


Figure C.13: Inter-server communication validation for a day of the week (Figure C.13a) and a day of the weekend (Figure C.13b) for two Websearch servers.

Validation of server interaction: Finally, we validate the accuracy of the model in capturing network interactions between specific servers. Potentially this is the highest source of inaccuracy for the model given its probabilistic nature. Figure C.13 shows the comparison between original and synthetic workload for the isolated network activity of a server pair across two different days (a weekday and a weekend day). The upper figures show traffic from Server A to Server B and the lower figures traffic from Server B to Server A. The two servers are chosen randomly across the pool of servers of the large Websearch cluster. As the graph shows, the hierarchical design and temporal variance of ECHO enables close resemblance for the point to point network traffic. The deviation between original and synthetic workloads is 12.2% for the weekday, on average and 8.9% for the weekend, which is higher than the system-wide deviation but still a very close approximation of the original traffic. Most of the error occurs when very low traffic in the original application is not reflected in the generated workload. We have verified the consistency of these results with different server pairs.

Overall, the network load captured and generated by the model deviates marginally from the original application in all four characteristics, with average deviation being smaller than the other metrics and the recreation of temporal patterns being slightly

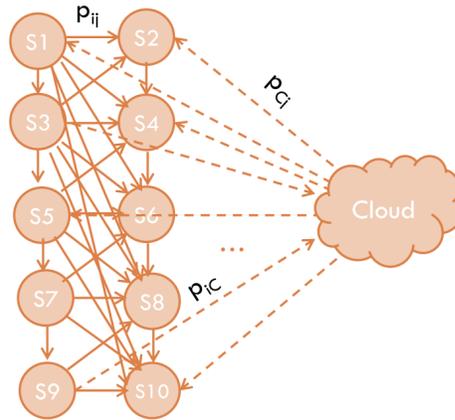


Figure C.14: Modified model for server subsets. The "Cloud" node represents traffic generated from or directed to external nodes in the system.

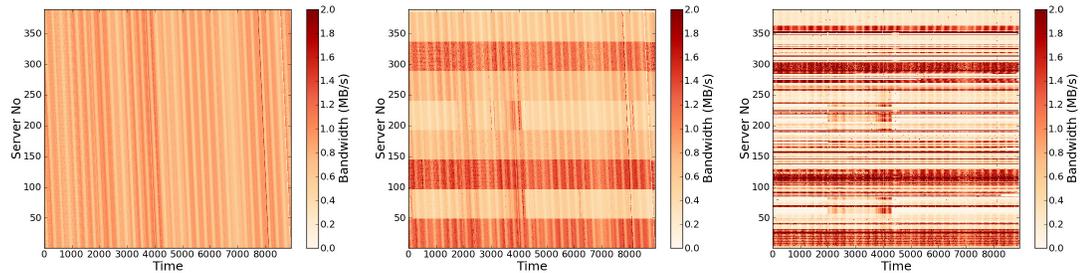


Figure C.15: Model representation when moving from single-level to two and three-level hierarchy for a DC running Combine.

more accurate than that of spatial patterns.

Sensitivity to model granularity: Finally, we examine the impact of hierarchy on the accuracy of the model. For the small DC running Combine we create three different sets of models; the first has information only at the granularity of groups of racks (level 1), the second at the granularity of individual racks (level 2) and the last one captures the workload at individual server granularity (level 3). Figure C.15 shows the network traffic generated by the model as more levels are added in the representation. Adding more levels significantly improves the accuracy of the model, as reflected in the corresponding deviations which are: 28% for the level-1, 9.1% for the level-2 and 4.4% for the level-3 model.

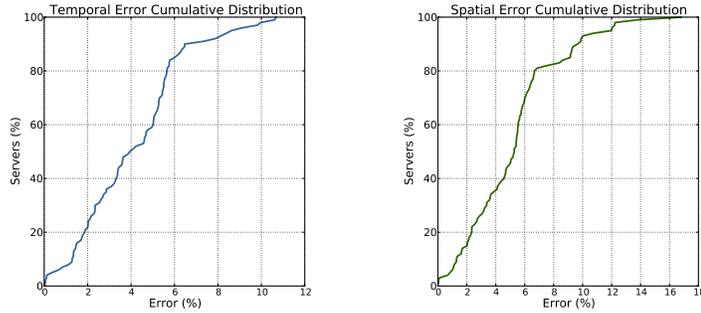


Figure C.16: CDFs of error distribution across servers for (a) network activity fluctuation and (b) network activity locality.

C.6 Conclusions

We presented ECHO, a concise and scalable modeling and generation scheme for network traffic in large-scale DCs. ECHO captures both the temporal and spatial characteristics of large-scale applications and recreates per-server and system-wide network traffic maps. ECHO is driven by robust analytical models which allow us to provide strong guarantees on the accuracy of the workload information. We have validated the accuracy of these models against real DC applications and have shown marginal deviations between original and generated traffic. Thus, ECHO can be used to derive confident decisions on DC-related system studies. Specifically, we have studied the temporal and spatial patterns of two large-scale production DCs running Websearch. ECHO can also be used for studies involving workload migration, and consolidation.

Bibliography

- [1] Controlling virtual machine sprawl: How to better utilize virtual infrastructure. In *White Paper*. November 2012. 17
- [2] *Migrate VMs with Zero Downtime*. <http://www.vmware.com/products/vmotion>. 26, 225, 251
- [3] Understanding memory resource management in vmware vsphere v. 5.0. In *Technical White Paper*. 2011. 17
- [4] Xenserver 4.0 <http://www.citrix.com/xenserver/>. 18, 24, 26, 57, 225
- [5] B. Abrahao and A. Zhang. Characterizing application workloads on cpu utilization for utility computing. Technical Report HPL-2004-157, HP Labs, 2004. 9
- [6] Adaptec maxiq. 32gb ssd cache performance kit. <http://www.adaptec.com/en-US/products/CloudComputing/MaxIQ/SSD-Cache-Performance/>. 221
- [7] Faraz Ahmad, Srimat T. Chakradhar, Anand Raghunathan, and T. N. Vijaykumar. Tarazu: optimizing mapreduce on heterogeneous clusters. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. London, UK, 2012. 82
- [8] Irfan Ahmad. Easy and efficient disk i/o workload characterization in vmware esx server. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*. Boston, MA, 2007. 204

- [9] Irfan Ahmad. Easy and efficient disk i/o workload characterization in vmware esx server. In *Proceedings of the 10th IEEE International Symposium on Workload Characterization*, pages 149–158, 2007. 9
- [10] Yasuhiro Ajiro and Atsuhiko Tanaka. Improving packing algorithms for server consolidation. In *Int. CMG Conference*, 2007. 253
- [11] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. In *Proceedings of the ACM SIGCOMM Conference on Data Communication*. Seattle, WA, 2008. 11
- [12] Alaa Alameldeen and David Wood. Ipc considered harmful for multiprocessor workloads. In *IEEE Micro*, July/Aug. 2006. 32
- [13] Amazon ec2. <http://aws.amazon.com/ec2/>. 9, 17, 18, 24, 44, 57, 165, 166, 225
- [14] Hrishikesh Amur, James Cipar, Varun Gupta, Gregory R. Ganger, Michael A. Kozuch, and Karsten Schwan. Robust and flexible power-proportional storage. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SOCC)*. Indianapolis, Indiana, USA, 2010. 227, 251, 252, 253
- [15] Ganesh Ananthanarayanan, Ali Ghodsi, Scott Shenker, and Ion Stoica. Effective straggler mitigation: Attack of the clones. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. Lombard, IL, 2013. 82
- [16] Ganesh Ananthanarayanan, Srikanth Kandula, Albert Greenberg, Ion Stoica, Yi Lu, Bikas Saha, and Edward Harris. Reining in the outliers in map-reduce clusters using mantri. In *Proceedings of the 9th USENIX conference on Operating Systems Design and Implementation (OSDI)*. Vancouver, CA, 2010. 82
- [17] Eric Anderson, Michael Hobbs, Kimberly Keeton, Susan Spence, Mustafa Uysal, and Alistair Veitch. Hippodrome: Running circles around storage administration. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies (FAST)*. Monterey, CA, 2002. 254

- [18] Eric Anderson, Susan Spence, Ram Swaminathan, Mahesh Kallahalla, and Qian Wang. Quickly finding near-optimal storage designs. *ACM Trans. Comput. Syst.*, 23(4):337–374, November 2005. 226, 254
- [19] Alexandr Andoni and Piotr Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In *Proceedings of the 47th IEEE Symposium on the Foundations of Computer Science (FOCS)*. Berkeley, CA, 2006. 147
- [20] Koushik Annapureddy. Security challenges in hybrid cloud infrastructures. In *Seminar on Network Security, Aalto University, T-110.5290*. 2010. 21, 197
- [21] Apache zookeeper. <http://zookeeper.apache.org/>. 83
- [22] Rami Atar, Avi Mandelbaum, and Martin Reiman. Scheduling a multi class queue with many exponential server: Asymptotic optimality in heavy traffic. In *Annals of Applied Probability, Vol. 14, No. 3*. 2004. 123, 133
- [23] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of SIGMETRICS*. London, UK, 2012. 9, 98, 159, 257, 261, 262
- [24] Autoscale. <https://cwiki.apache.org/cloudstack/autoscaling.html>. 57, 85, 176
- [25] Aws autoscaling. <http://aws.amazon.com/autoscaling/>. 85, 176
- [26] Gaurav Banga, Peter Druschel, and Jeffrey Mogul. Resource containers: a new facility for resource management in server systems. In *Proceedings of OSDI*. New Orleans, LA, 1999. 57, 82, 170
- [27] Paul Barford and Mark Crovella. Generating representative web workloads for network and server performance evaluation. In *Proceedings of the 1998 ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*. Madison, Wisconsin, USA, 1998. 257, 259

- [28] Sean Kenneth Barker and Prashant Shenoy. Empirical evaluation of latency-sensitive application performance in the cloud. In *Proceedings of MMsys*. Scottsdale, AR, 2010. [166](#)
- [29] Luiz Barroso. Warehouse-scale computing: Entering the teenage decade. *ISCA Keynote, SJ, June 2011*. [12](#), [25](#), [60](#), [62](#), [165](#), [173](#), [225](#)
- [30] Luiz Barroso, Jeffrey Dean, and Urs Hoelzle. Web search for a planet: The google cluster architecture. In *Journal IEEE Micro archive, Volume 23 Issue 2, page 22-28, March 2003*. [9](#), [10](#)
- [31] Luiz Barroso and Urs Hoelzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. MC Publishers, 2009. [1](#), [2](#), [9](#), [12](#), [13](#), [24](#), [25](#), [109](#), [119](#), [165](#), [170](#), [173](#), [174](#), [225](#), [226](#), [238](#), [239](#)
- [32] Luiz Barroso and Urs Hölzle. The case for energy-proportional computing. *Computer*, 40(12):33–37, December 2007. [12](#)
- [33] Novella Bartolini, Giancarlo Bongiovanni, and Simone Silvestri. Self-overload control for distributed web systems. In *Proceedings of the International Workshop on Quality of Service (IWQoS)*. Enschede, 2008. [131](#)
- [34] Robert Bell, Yehuda Koren, and Chris Volinsky. The bellkor 2008 solution to the netflix prize. Technical report, 2007. [26](#), [28](#), [67](#), [70](#), [116](#)
- [35] Anton Beloglazov and Rajkumar Buyya. Managing overloaded hosts for dynamic consolidation of virtual machines in cloud data centers under quality of service constraints. *IEEE Trans. Parallel Distrib. Syst.*, 24(7):1366–1379, July 2013. [227](#)
- [36] Orna Agmon Ben-Yehuda, Muli Ben-Yehuda, Assaf Schuster, and Dan Tsafir. Deconstructing amazon ec2 spot instance pricing. In *ACM TEAC, 1(3), September 2013*. [21](#), [197](#)
- [37] Theophilus Benson, Aditya Akella, and David A. Maltz. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM*

- Conference on Internet Measurement (IMC)*. Melbourne, Australia, 2010. [257](#), [261](#)
- [38] Theophilus Benson, Ashok Anand, Aditya Akella, and Ming Zhang. Microcrote: Fine grained traffic engineering for data centers. In *Proceedings of the Seventh Conference on Emerging Networking EXperiments and Technologies (CoNEXT)*. Tokyo, Japan, 2011. [257](#)
- [39] Dimitris Bertsimas, David Gamarnik, and John Tsitsiklis. Performance of multiclass markovian queueing networks via piecewise linear lyapunov functions. In *Annals of Applied Probability, Vol. 00, No. 0, 1-45*. 2001. [123](#), [132](#)
- [40] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. Toronto, CA, October, 2008. [45](#), [84](#), [95](#), [110](#), [126](#), [160](#)
- [41] Ramazan Bitirgen, Engin Ipek, and Jose F. Martinez. Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach. In *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Lake Como, Italy, 2008. [22](#)
- [42] Peter Bodik, Armando Fox, Michael J. Franklin, Michael I. Jordan, and David A. Patterson. Characterizing, modeling, and generating workload spikes for stateful services. In *Proceedings of SOCC*, pages 241–252, 2010. [9](#)
- [43] Leon Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of the International Conference on Computational Statistics (COMPSTAT)*. Paris, France, 2010. [30](#), [70](#), [115](#), [139](#)
- [44] Gerd Breiter and Vijay K. Naik. A framework for controlling and managing hybrid cloud service integration. In *Proceedings of IC2E*. Redwood City, CA, 2013. [21](#), [197](#)

- [45] Brad Calder, Ju Wang, et al. Windows azure storage: a highly available cloud storage service with strong consistency. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*. Cascais, Portugal, 2011. 9, 26
- [46] Maria Calzarossa and Giuseppe Serazzi. Workload Characterization: a Survey. *Proceedings of the IEEE*, 8(81):1136–1150, 1993. 257
- [47] Jakob Carlström and Raphael Rom. Application-aware admission control and scheduling in web servers. In *Proceedings of Infocom*. New York, NY, 2002. 120, 131
- [48] Marcus Carvalho, Walfredo Cirne, Francisco Brasileiro, and John Wilkes. Long-term slos for reclaimed cloud computing resources. In *Proceedings of SOCC*. Seattle, WA, 2014. 17, 174
- [49] Apache cassandra. <http://cassandra.apache.org/>. 10, 72, 85, 160
- [50] A. Chandra, W. Gong, and P. Shenoy. Dynamic resource allocation for shared data centers using online measurements. In *Proceedings of International Workshop on Quality of Service (IWQoS)*. Montreal, CA, 2004. 16
- [51] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Seattle, WA, 2006. 10, 251
- [52] Hyeong Soo Chang, Robert Givan, and Edwin Chong. On-line scheduling via sampling. In *Proceedings of Artificial Intelligence Planning and Scheduling (AIPS)*. 2000. 20, 138
- [53] Moses S. Charikar. Similarity estimation techniques from rounding algorithms. In *Proceedings of the 34th Annual ACM Symposium on Theory of Computing 2002*. 147

- [54] Jeffrey Chase, Darrell Anderson, Prachi Thakar, Amin Vahdat, and Ronald Doyle. Managing energy and server resources in hosting centers. In *Proceedings of SOSP*. Banff, CA, 2001. [16](#), [57](#), [176](#), [225](#), [252](#)
- [55] Gong Chen, Wenbo He, Jie Liu, Suman Nath, Leonidas Rigas, Lin Xiao, and Feng Zhao. Energy-aware server provisioning and load dispatching for connection-intensive internet services. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. San Francisco, California, 2008. [225](#)
- [56] Hong Chen. Fluid approximations and stability of multiclass queueing networks: Work-conserving disciplines. In *Annals of Applied Probability, Vol. 5, No. 3*. 1995. [132](#)
- [57] Yanpei Chen, Archana Ganapathi, Rean Griffith, and Randy Katz. The case for evaluating mapreduce performance using workload suites. In *Proceedings of the 2011 IEEE 19th Annual International Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 390–399, 2011. [9](#)
- [58] Sheng-Tzong Cheng, Chi-Ming Chen, and Ing-Ray Chen. Performance evaluation of an admission control algorithm: dynamic threshold with negotiation. In *Performance Evaluation, vol. 52*. 2003. [132](#)
- [59] Ludmila Cherkasova and Pete Phaal. Predictive admission control strategy for overloaded commercial web server. In *Proceedings of the international symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*. San Francisco, CA, 2000. [131](#), [132](#)
- [60] Ludmila Cherkasova and Peter Phaal. Session-based admission control: A mechanism for peak load management of commercial web sites. In *IEEE Transactions on Computers, Vol. 51, No. 6*. 2002. [131](#)
- [61] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual

- machines. In *Proceedings of the Second Conference on Symposium on Networked Systems Design & Implementation (NSDI)*. Boston, MA, 2005. 16, 225
- [62] Cluster management: Maintenance and monitoring. <http://www.cloudera.com/content/cloudera/en/developers/home/developer-admin-resources/cluster-management.html>. 14
- [63] McKinsey & Company. Revolutionizing data center efficiency. In *Uptime Institute Symposium, 2008*. 61, 66
- [64] Computing Community Consortium (CCC). "21st century computer architecture: A community white paper". <http://www.cccb.org/2012/05/29/21st-century-computer-architecture>, May 2012. 11
- [65] Computing Community Consortium (CCC). Challenges and opportunities with big data. <http://cra.org/ccc/docs/init/bigdatawhitepaper.pdf>, February 2012. 9
- [66] Linux containers. <http://lxc.sourceforge.net/>. 141, 170
- [67] Peter B. Danzig, Sugih Jamin, Ramres, Danny J. Mitzel, and Deborah Estrin. An empirical workload model for driving wide-area tcp/ip network simulations. *Internetworking: Research and Experience*, 3:1–26, 1992. 259
- [68] Jeffrey Dean and Luiz Andre Barroso. The tail at scale. In *CACM, Vol. 56 No. 2, Pages 74-80*. 10, 62, 81, 167, 173
- [69] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. In *Proceedings of OSDI*, pages 10–10, 2004. 9, 10, 82, 98, 201, 212, 245
- [70] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 205–220, 2007. 9

- [71] Ewa Deelman, Gurmeet Singh, Miron Livny, Bruce Berriman, and John Good. The cost of doing science on the cloud: The montage example. In *Proceedings of SC*. Austin, TX, 2008. 20, 197
- [72] Christina Delimitrou, Nick Bambos, and Christos Kozyrakis. QoS-Aware Admission Control in Heterogeneous Datacenters. In *Proceedings of the International Conference of Autonomic Computing (ICAC)*. San Jose, CA, USA, 2013. 6, 86
- [73] Christina Delimitrou and Christos Kozyrakis. Amdahl's Law for Latency-Critical Applications. <http://www.stanford.edu/~cdel/2014.amdahls.slides.pdf>, November 2014. 3
- [74] Christina Delimitrou and Christos Kozyrakis. iBench: Quantifying Interference for Datacenter Workloads. In *Proceedings of the 2013 IEEE International Symposium on Workload Characterization (IISWC)*. Portland, OR, September 2013. 4, 74, 141
- [75] Christina Delimitrou and Christos Kozyrakis. Optimizing Resource Provisioning in Shared Cloud Systems. *in submission*. 7
- [76] Christina Delimitrou and Christos Kozyrakis. Paragon: QoS-Aware Scheduling for Heterogeneous Datacenters. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Houston, TX, USA, 2013. 4, 20, 62, 65, 69, 71, 96, 98, 109, 116, 119, 128, 135, 136, 137, 139, 140, 141, 196
- [77] Christina Delimitrou and Christos Kozyrakis. QoS-Aware Scheduling in Heterogeneous Datacenters with Paragon. In *ACM Transactions on Computer Systems (TOCS)*, Vol. 31 Issue 4. December 2013. 4
- [78] Christina Delimitrou and Christos Kozyrakis. Quality-of-Service-Aware Scheduling in Heterogeneous Datacenters with Paragon. In *IEEE Micro Special Issue on Top Picks from the Computer Architecture Conferences*. May/June 2014. 4

- [79] Christina Delimitrou and Christos Kozyrakis. Quasar: Resource-Efficient and QoS-Aware Cluster Management. In *Proceedings of the Nineteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Salt Lake City, UT, USA, 2014. [2](#), [5](#), [12](#), [19](#), [135](#), [137](#), [138](#), [139](#), [140](#), [141](#), [156](#), [168](#), [170](#), [174](#), [175](#), [176](#)
- [80] Christina Delimitrou and Christos Kozyrakis. The Netflix Challenge: Data-center Edition. Los Alamitos, CA, USA, July 2012. IEEE Computer Society. [4](#)
- [81] Christina Delimitrou, Daniel Sanchez, and Christos Kozyrakis. Tarcil: Reconciling Scheduling Speed and Quality in Large Shared Clusters. In *Proceedings of the Sixth ACM Symposium on Cloud Computing (SOCC)*. Kohala Coast, HI, 2015. [6](#)
- [82] Christina Delimitrou, Daniel Sanchez, and Christos Kozyrakis. Tarcil: Reconciling Scheduling Speed and Quality in Large Shared Clusters. *in submission*. [6](#)
- [83] Christina Delimitrou, Sriram Sankar, Aman Kansal, and Christos Kozyrakis. ECHO: Recreating Network Traffic Maps for Datacenters of Tens of Thousands of Servers. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*. San Diego, CA, USA, 2012. [7](#), [98](#)
- [84] Christina Delimitrou, Sriram Sankar, Badriddine Khessib, Kushagra Vaid, and Christos Kozyrakis. Time and Cost-Efficient Modeling and Generation of Large-Scale TPC Workloads. In *Proceedings of the Third TPC Technology Conference on Performance Evaluation & Benchmarking (TPC TC)*. In conjunction with *VLDB*. Seattle, WA, USA, 2011. [7](#), [98](#), [238](#), [239](#)
- [85] Christina Delimitrou, Sriram Sankar, Kushagra Vaid, and Christos Kozyrakis. Decoupling Datacenter Studies from Access to Large-Scale Applications: A

- Modeling Approach for Storage Workloads. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*. Austin, TX, USA, 2011. 7, 9, 98, 226, 231, 232, 238, 239
- [86] Christina Delimitrou, Sriram Sankar, Kushagra Vaid, and Christos Kozyrakis. Storage I/O Generation and Replay for Datacenter Applications. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. Austin, TX, USA, 2011. 7, 98
- [87] Christina Delimitrou, Sriram Sankar, Kushagra Vaid, and Christos Kozyrakis. Decoupling Datacenter Studies from Access to Large-Scale Applications: A Modeling Approach for Storage Workloads. In *IEEE Computer Architecture Letters*, Los Alamitos, CA, USA, January-June 2012. 7, 98
- [88] Christina Delimitrou, Sriram Sankar, Kushagra Vaid, and Christos Kozyrakis. Accurate Modeling and Generation of Storage I/O for Datacenter Workloads. In *Proceedings of the 2nd Exascale Evaluation and Research Techniques Workshop (EXERT), in conjunction with ASPLOS*, Newport Beach, CA, USA, 2011. 7, 98, 238
- [89] S. Di, D. Kundo, and W. Cirne. Characterization and comparison of google cloud load versus grids. <http://hal.archives-ouvertes.fr/hal-00705858>, 2012. 9
- [90] Diskspd: File and network i/o using win32 and .net api's on windows xp. <http://research.microsoft.com/en-us/um/siliconvalley/projects/sequentialio/>. 205, 208
- [91] Xicheng Dong, Ying Wang, and Huaming Liao. Scheduling mixed real-time and non-real-time applications in mapreduce environment. In *Proceedings of the IEEE 17th International Conference on Parallel and Distributed Systems (ICPADS)*. Tainan, 2011. 20, 138
- [92] Allen B. Downey and Dror G. Feitelson. The elusive goal of workload characterization. *SIGMETRICS Perform. Eval. Rev.*, 26(4):14–29, March 1999. 257

- [93] R. Doyle, J. Chase, O. Asad, W. Jin, and A. Vahdat. Model-based resource provisioning in a web service utility. In *Proceedings of the 4th conference on USENIX Symposium on Internet Technologies and Systems (USITS)*. Seattle, WA, 2003. 16
- [94] Cliff Engle, Antonio Lupher, Reynold Xin, Matei Zaharia, Michael J. Franklin, Scott Shenker, and Ion Stoica. Shark: Fast data analysis using coarse-grained distributed memory. In *Proceedings of ACM SIGMOD*. Scottsdale, Arizona, 2012. 157
- [95] Deniz Ersoz, Mazin S. Yousif, and Chita R. Das. Characterizing network traffic in a cluster-based, multi-tier data center. In *Proceedings of ICDCS*. Toronto, Canada, 2007. 257, 261
- [96] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *Proceedings of the 38th annual Intl. symposium on Computer architecture (ISCA)*. San Jose, California, USA, 2011. 11
- [97] Etw: Event tracing for windows. <http://msdn.microsoft.com/en-us/library/bb968803%28VS.85%29.aspx>. 203, 213, 222
- [98] Eucalyptus cloud services. <http://www.eucalyptus.com/>. 62
- [99] Xiaobo Fan, Wolf-Dietrich Weber, and Luiz Andre Barroso. Power provisioning for a warehouse-sized computer. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*. San Diego, CA, 2007. 16, 225, 239
- [100] Benjamin Farley, Ari Juels, Venkatanathan Varadarajan, Thomas Ristenpart, Kevin D. Bowers, and Michael M. Swift. More for your money: Exploiting performance heterogeneity in public clouds. In *Proceedings of SOCC*. San Jose, CA, 2012. 166, 174
- [101] Alexandra Fedorova, Margo Seltzer, and Michael D. Smith. Improving performance isolation on chip multiprocessors via an operating system scheduler. In

- Proceedings of the 16th Intl. Conference on Parallel Architecture and Compilation Techniques (PACT)*. Brasov, Romania, 2007. 58
- [102] Alexandra Fedorova, David Vengerov, and Daniel Doucette. Operating system on heterogeneous core systems. In *Proceedings of Operating System Support for Heterogeneous Multicore Architectures (OSHMA)*, 2007. 18, 58
- [103] Dror Feitelson and Bill Nitzberg. Job characteristics of a production parallel scientific workload on the nasa ames ipsc/860. In *Proceedings of JSSPP*. 1995. 257
- [104] Dror G. Feitelson. Metric and workload effects on computer systems evaluation. *Computer*, 36(9):18–25, September 2003. 257, 258, 261
- [105] Yuan Feng, Baochun Li, and Bo Li 0001. Peer-to-peer bargaining in container-based datacenters. In *Proceedings of IPTPS*. San Jose, CA, 2010. 252
- [106] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafae, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. Clearing the clouds: A study of emerging scale-out workloads on modern hardware. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. London, England, UK, 2012. 98
- [107] Brad Fitzpatrick. Distributed caching with memcached. In *Linux Journal, Volume 2004, Issue 124, 2004*. 10, 15, 85, 97, 106, 112
- [108] Rohan Gandhi and Amit Sabne. Finding stragglers in hadoop. In *Technical Report*. 2011. 82
- [109] Gartner says efficient data center design can lead to 300 percent capacity growth in 60 percent less space. <http://www.gartner.com/newsroom/id/1472714>. 61, 66
- [110] Google compute engine. <https://developers.google.com/compute/>. 9, 24, 44, 165, 166, 168, 171

- [111] Orhan Gemikonakli, Enver Ever, and Eser Gemikonakli. Performance modeling of virtualized servers. In *Proceedings of Computer Modelling and Simulation Conference (UKSim)*. Cambridge, UK, 2010. [133](#)
- [112] Zoubin Ghahramani and Michael Jordan. Learning from incomplete data. In *Lab Memo No. 1509, CBCL Paper No. 108, MIT AI Lab*. [74](#)
- [113] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of SOSP, New York, NY, 2003*. [253](#)
- [114] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant resource fairness: fair allocation of multiple resource types. In *Proceedings of the 8th USENIX conference on Networked systems design and implementation (NSDI)*. Boston, MA, 2011. [17](#), [65](#)
- [115] A. Gidenstam, B. Koldehofe, M. Papatriantafilou, and P. Tsigas. Dynamic and fault-tolerant cluster management. In *Proc. of the Fifth IEEE International Conference on Peer-to-Peer Computing*. Konstanz, Germany. [13](#)
- [116] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. Understanding network failures in data centers: Measurement, analysis, and implications. In *Proceedings of the ACM SIGCOMM 2011 Conference*, pages 350–361. Toronto, Ontario, Canada, 2011. [258](#)
- [117] Daniel Gmach, Jerry Rolia, Ludmila Cherkasova, and Alfons Kemper. Workload analysis and demand prediction of enterprise data center applications. In *Proceedings of IISWC*. Boston, MA, 2007. [17](#), [57](#), [65](#), [196](#)
- [118] Zhenhuan Gong, Xiaohui Gu, and John Wilkes. Press: Predictive elastic resource scaling for cloud systems. In *Proceedings of CNSM*. Niagara Falls, ON, 2010. [16](#), [65](#), [81](#), [196](#)
- [119] Sriram Govindan, Jeonghwan Choi, Bhuvan Urgaonkar, Anand Sivasubramanian, and Andrea Baldini. Statistical profiling-based techniques for effective

- power provisioning in data centers. In *Proceedings of EuroSys*. Nuremberg, Germany, 2009. 16
- [120] Sriram Govindan, Jie Liu, Aman Kansal, and Anand Sivasubramaniam. Cuanta: quantifying effects of shared on-chip resource interference for consolidated virtual machines. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*. Cascais, Portugal, 2011. 13, 17, 20, 26, 56, 98, 135, 139
- [121] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. Multi-resource packing for cluster schedulers. In *Proc. of the 2014 ACM Conference on SIGCOMM*. Chicago, Illinois, 2014. 20
- [122] Albert Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta. V12: a scalable and flexible data center network. In *Proceedings of the ACM SIGCOMM Conference on Data communication*. Barcelona, Spain, 2009. 11
- [123] Geoffrey R. Grimmett and David R. Stirzaker. Probability and random processes. 2nd Edition. Clarendon Press, Oxford, 1992. 180
- [124] B. Guenter, N. Jain, and C. Williams. Managing cost, performance, and reliability tradeoffs for energy-aware server provisioning. In *Proceedings of the IEEE International Conference on Computer Communications (INFOCOM)*. Shanghai, China, 2011. 16
- [125] Jorge Guerra, Wendy Belluomini, Joseph Glider, Karan Gupta, and Himabindu Pucha. Energy proportionality for storage: Impact and feasibility. *SIGOPS Oper. Syst. Rev.*, 44(1):35–39, March 2010. 252
- [126] Jorge Guerra, Himabindu Pucha, Joseph Glider, Wendy Belluomini, and Raju Rangaswami. Cost effective storage using extent based dynamic tiering. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies (FAST)*. San Jose, CA, 2011. 225, 252

- [127] Marisabel Guevara, Benjamin Lubin, and Benjamin C. Lee. Navigating heterogeneous processors with market mechanisms. In *Proceedings of HPCA*. Shenzhen, China, 2013. 21, 197
- [128] Jordi Guitart, David Carrera, Vicenc Beltran, Jordi Torres, and Eduard Ayguade. Designing an overload control strategy for secure e-commerce applications. In *Computer Networks, vol. 51*. 2007. 132
- [129] Ajay Gulati, Chethan Kumar, and Irfan Ahmad. Modeling workloads and devices for io load balancing in virtualized environments. 37(3):61–66, January 2010. 9
- [130] Ajay Gulati, Chethan Kumar, Irfan Ahmad, and Karan Kumar. Basil: Automated io load balancing across storage devices. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies (FAST)*, pages 13–13. San Jose, California, 2010. 225, 251, 253
- [131] Ajay Gulati, Ganesha Shanmuganathan, Irfan Ahmad, Carl A. Waldspurger, and Mustafa Uysal. Pesto: online storage performance management in virtualized datacenters. In *Proceedings of the Symposium on Cloud Computing (SoCC)*. Cascais, Portugal, 2011. 253
- [132] Rohit Gupta, Sumit Kumar Bose, Srikanth Sundarrajan, Manogna Chebiyam, and Anirban Chakrabarti. A two stage heuristic algorithm for solving the server consolidation problem with item-item and bin-item incompatibility constraints. In *IEEE SCC (2)*, pages 39–46. IEEE Computer Society, 2008. 252, 253
- [133] Itai Gurvich. Design and control of the m/m/n queue with multi-type customers and many servers. In *M.S. Thesis, Technion Israel Institute of Technology*. 2004. 133
- [134] Apache hadoop. <http://hadoop.apache.org/>. 84, 97, 98
- [135] Greg Hamerly, Erez Perelman, Jeremy Lau, Brad Calder, and Timothy Sherwood. Using machine learning to guide architecture simulation. *Journal of Machine Learning Research*, 7:343–378, 2006. 22

- [136] James Hamilton. Cost of power in large-scale data centers. <http://perspectives.mvdirona.com>. 1, 10, 11, 13, 25, 165
- [137] James Hamilton. Internet-scale service infrastructure efficiency. In *Proceedings of the 37th Intl. Symposium on Computer architecture*, Austin, TX, 2009. 11, 13
- [138] John Jay Hasenbein. Stability, capacity and scheduling of multiclass queueing networks. In *Ph.D. Thesis. Georgia Institute of Technology*. 1998. 123
- [139] Ben Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of NSDI*. Boston, MA, 2011. 13, 14, 19, 26, 57, 60, 62, 65, 72, 135, 138, 149, 165, 196
- [140] Storage consolidation for growing environments. Hitachi White Paper. April 2008. 225, 252
- [141] Henry Hoffmann, Stelios Sidiropoulos, Michael Carbin, Sasa Misailovic, Anant Agarwal, and Martin Rinard. Dynamic knobs for responsive power-aware computing. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Newport Beach, CA, USA, 2011. 225, 252
- [142] Mark Horowitz. Scaling, power and the future of cmos. In *Proceedings of the 20th Intl. Conference on VLSI Design held jointly with 6th International Conference: Embedded Systems, VLSID '07*, 2007. 11
- [143] Reza Hoseinyfarahabady, Hamid Samani, Luke Leslie, Young Choon Lee, and Albert Zomaya. Handling uncertainty: Pareto-efficient bot scheduling on hybrid clouds. In *Proceedings of ICPP*. Lyon, France, 2013. 21, 197
- [144] Hotcrp conference management system. <http://read.seas.harvard.edu/~kohler/hotcrp/>. 85

- [145] Lan Huang, Gang Peng, and Tzi-cker Chiueh. Multi-dimensional storage virtualization. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*. New York, NY, USA, 2004. 225, 253
- [146] Ibm websphere: Security configuration properties. https://www-01.ibm.com/support/knowledgecenter/SSFPJS_8.5.0/com.ibm.wbpm.admin.doc/topics/csec_config_properties.html. 14
- [147] Iometer: performance analysis tool. <http://www.iometer.org/>. 204, 218
- [148] Alexandru Iosup, Nezhir Yigitbasi, and Dick Epema. On the performance variability of production cloud services. In *Proceedings of CCGRID*. Newport Beach, CA, 2011. 166
- [149] 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 Symposium on Operating Systems Principles (SOSP)*. Big Sky, MT, 2009. 18, 19, 20, 135, 149
- [150] Aamer Jaleel, Matthew Mattina, and Bruce L. Jacob. Last level cache (llc) performance of data mining workloads on a cmp - a case study of parallel bioinformatics workloads. In *Proceedings of the 12th International Symposium on High-Performance Computer Architecture (HPCA-12)*. Austin, Texas, 2006. 45, 84, 95, 110, 126
- [151] Youngmi Joo, Vinay Ribeiro, Anja Feldmann, Anna C. Gilbert, and Walter Willinger. Tcp/ip traffic dynamics and network performance: A lesson in workload modeling, flow control, and trace-driven simulations. *SIGCOMM Comput. Commun. Rev.*, 31(2):25–37, April 2001. 98, 259
- [152] Sukhun Kang and Rakesh Kumar. Magellan: A search and machine learning-based framework for fast multi-core design space exploration and optimization. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*. Munich, Germany, 2008. 22

- [153] Jonathan Katz and Yehuda Lindell. Introduction to modern cryptography. *Chapman & Hall/CRC Press, 2007*. [30](#), [40](#)
- [154] Rini T. Kaushik and Milind Bhandarkar. Greenhdfs: Towards an energy-conserving, storage-efficient, hybrid hadoop compute cluster. In *Proceedings of the International Conference on Power Aware Computing and Systems (HotPower)*. Vancouver, BC, Canada, 2010. [253](#)
- [155] Swaroop Kavalanekar, Dushyanth Narayanan, Sriram Sankar, Eno Thereska, Kushagra Vaid, and Bruce Worthington. Measuring database performance in online services: a trace-based approach. In *Proceedings of TPC TC*. Lyon, France, 2009. [204](#)
- [156] Swaroop Kavalanekar, Bruce Worthington, Qi Zha, and Vishal Sharda. Characterization of storage workload traces from production windows servers. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*. Seattle, WA, 2008. [9](#), [204](#), [210](#)
- [157] Soila Kavulya, Jiaqi Tan, Rajeev Gandhi, and Priya Narasimhan. An analysis of traces from a production mapreduce cluster. In *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, pages 94–103, 2010. [9](#)
- [158] Steve Keckler. Life after dennard and how i learned to love the picojoule. Keynote at the he 44th Intl. Symposium on Microarchitecture, December 2011. [11](#)
- [159] Vaibhav Khadilkar, Kerim Yasin Oktay, Bijit Hore, Murat Kantarcioglu, Sharad Mehrotra, and Bhavani Thuraisingham. Risk-aware data processing in hybrid clouds. TR-UTDCS-31-11, 2011. [21](#), [197](#)
- [160] Yaakoub El Khamra, Hyunjoo Kim, Shantenu Jha, and Manish Parashar. Exploring the performance fluctuations of hpc workloads on clouds. In *Proceedings of CloudCom*. Indianapolis, IN, 2010. [166](#)

- [161] Krzysztof C. Kiwiel. Convergence and efficiency of subgradient methods for quasiconvex minimization. In *Mathematical Programming (Series A) (Berlin, Heidelberg: Springer) 90 (1): pp. 1-25, 2001.* [30](#), [70](#), [74](#), [139](#)
- [162] Leonard Kleinrock. Queueing systems volume 1: Theory. pp. 101-103, 404. [173](#)
- [163] Christos Kozyrakis, Aman Kansal, Sriram Sankar, and Kushagra Vaid. Server engineering insights for large-scale online services. *IEEE Micro, vol.30, no.4, July 2010.* [13](#), [25](#), [119](#), [151](#), [203](#), [226](#), [238](#), [252](#)
- [164] Vidyadhar G. Kulkarni and Natarajan Gautam. Admission control of multi-class traffic with service priorities in high-speed networks. In *Journal of Queueing Systems: Theory and Applications archive Vol. 27, No. 1/2.* 1997. [123](#), [132](#)
- [165] Dara Kusic, Jeffrey O. Kephart, James E. Hanson, Nagarajan Kandasamy, and Guofei Jiang. Power and performance management of virtualized computing environments via lookahead control. *Cluster Computing*, 12(1):1–15, 2009. [227](#)
- [166] Eric Lau, Jason E Miller, Inseok Choi, Donald Yeung, Saman Amarasinghe, and Anant Agarwal. Multicore performance optimization using partner cores. In *Proceedings of HotPar*. Berkeley, CA, 2011. [168](#)
- [167] James Laudon. Performance/watt: the new server focus. In *ACM SIGARCH Computer Architecture News: dasCMP*. Vol. 33 Issue 4, p. 5-13, November 2005. [168](#)
- [168] Sangmin Lee, Rina Panigrahy, Vijayan Prabhakaran, Venugopalan Ramasubramanian, Kunal Talwar, Lincoln Uyeda, and Udi Wieder. Validating heuristics for virtual machines consolidation. Technical Report MSR-TR-2011-9, January 2011. [253](#)
- [169] Jacob Leverich and Christos Kozyrakis. On the energy (in)efficiency of hadoop clusters. In *Proceedings of HotPower*. Big Sky, MT, 2009. [61](#)

- [170] Jacob Leverich and Christos Kozyrakis. Reconciling high server utilization and sub-millisecond quality-of-service. In *Proceedings of EuroSys*. Amsterdam, The Netherlands, 2014. 12, 160
- [171] Jacob Leverich and Christos Kozyrakis. On the energy (in)efficiency of hadoop clusters. *SIGOPS Oper. Syst. Rev.*, 44(1):61–65, March 2010. 25, 253
- [172] Ang Li, Xiaowei Yang, Srikanth Kandula, and Ming Zhang. Cloudcmp: Comparing public cloud providers. In *Proceedings of IMC*. Melbourne, Australia, 2010. 20, 166, 197
- [173] Hui Li. Realistic workload modeling and its performance impacts in large-scale science grids. *IEEE Transactions on Parallel and Distributed Systems*, 21(4):480–493, 2010. 258, 260
- [174] M. Li, D. Goldberg, W. Tao, and Y. Tamir. Fault-tolerant cluster management for reliable high performance computing. In *Proc. of the 13th International Conference on Parallel and Distributed Computing and Systems*. Anaheim, CA, 2001. 13
- [175] Ching-Chi Lin, Pangfeng Liu, and Jan-Jan Wu. Energy-aware virtual machine dynamic provision and scheduling for cloud computing. In *Proceedings of the 2011 IEEE 4th International Conference on Cloud Computing (CLOUD)*. Washington, DC, USA, 2011. 227
- [176] Jimmy Lin. The curse of zipf and limits to parallelization: A look at the stragglers problem in mapreduce. In *Proceedings of LSDS-IR Workshop*. Boston, MA, 2009. 82
- [177] Host server cpu utilization in amazon ec2 cloud. <http://huanliu.wordpress.com/2012/02/17/host-server-cpu-utilization-in-amazon-ec2-cloud/>. 2, 61, 66, 174
- [178] Xue Liu, Jin Heo, Lui Sha, and Xiaoyun Zhu. Adaptive control of multi-tiered web applications using queueing predictor. In *IEEE Transactions on Network and Service Management*. September 2008. 131, 132

- [179] Zitao Liu and Sangyeun Cho. Characterizing machines and workloads on a google cluster. In *Parallel Processing Workshops (ICPPW), 2012 41st International Conference on*, pages 397–403, sept. 2012. 9
- [180] Jacob Lorch and Alan Smith. Reducing processor power consumption by improving processor time management in a single-user operating system. In *Proceedings of MOBICOM*. New York, NY, 1996. 173
- [181] Chenyang Lu, Guillermo A. Alvarez, and John Wilkes. Aqueduct: Online data migration with performance guarantees. In *Proceedings of File and Storage Technologies (FAST)*. Monterey, CA, 2002. 225, 253
- [182] Apache lucene. <http://lucene.apache.org/core/>. 98
- [183] Priya Mahadevan, Puneet Sharma, Sujata Banerjee, and Parthasarathy Ranganathan. Energy aware network operations. In *Proceedings of the 28th IEEE International Conference on Computer Communications Workshops (INFOCOM)*. Rio de Janeiro, Brazil, 2009. 225
- [184] Mahout. <http://mahout.apache.org/>. 84, 87, 166, 169
- [185] Dave Mangot. Ec2 variability: The numbers revealed. http://tech.mangot.com/roller/dave/entry/ec2_variability_the_numbers_revealed. 166
- [186] Jason Mars and Lingjia Tang. Whare-map: heterogeneity in "homogeneous" warehouse-scale computers. In *Proceedings of ISCA*. Tel-Aviv, Israel, 2013. 13, 17, 20, 66, 135, 139, 141, 168, 196
- [187] Jason Mars, Lingjia Tang, and Robert Hundt. Heterogeneity in "homogeneous"; warehouse-scale computers: A performance opportunity. *IEEE Comput. Archit. Lett.*, 10(2), July 2011. 17, 26, 28, 35, 56, 96, 98, 99
- [188] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. Bubble-up: increasing utilization in modern warehouse scale computers via sensible co-locations. In *Proceedings of MICRO*. Porto Alegre, Brazil, 2011. 13, 25, 26, 56, 62, 65, 196

- [189] Steven M. Martin, Krisztian Flautner, Trevor Mudge, and David Blaauw. Combined dynamic voltage scaling and adaptive body biasing for lower power microprocessors under dynamic workloads. In *Proceedings of ICCAD*. 2002. 173
- [190] José F. Martínez and Engin Ipek. Dynamic multicore resource management: A machine learning approach. *IEEE Micro*, 29(5):8–17, 2009. 22
- [191] David Meisner, Christopher M. Sadler, Luiz André Barroso, Wolf-Dietrich Weber, and Thomas F. Wenisch. Power management of online data-intensive services. In *Proceedings of the 38th annual international symposium on Computer architecture*, pages 319–330, 2011. 12, 25, 61, 109, 225
- [192] Bruce L. Miller. A queueing reward system with several customer classes. In *Management Science*, 16(3):234–245. 1969. 123, 132
- [193] Michael Mitzenmacher. The power of two choices in randomized load balancing. In *Journal IEEE Transactions on Parallel and Distributed Systems, Volume 12 Issue 10, 2001*. 20, 138
- [194] Consolidating backup infrastructures. <http://www.microsoft.com/windowsserversystem/storage/consolidation.aspx>. 225, 252
- [195] Dushyanth Narayanan, Austin Donnelly, and Antony Rowstron. Write off-loading: Practical power management for enterprise storage. *Trans. Storage*, 4(3):10:1–10:23, November 2008. 238, 239, 252
- [196] Dushyanth Narayanan, Austin Donnelly, Eno Thereska, Sameh Elnikety, and Antony Rowstron. Everest: Scaling down peak loads through i/o off-loading. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI)*. San Diego, California, 2008. 252
- [197] Dushyanth Narayanan, Eno Thereska, Austin Donnelly, Sameh Elnikety, and Antony Rowstron. Migrating server storage to ssds: Analysis of tradeoffs. In *Proceedings of the 4th ACM European Conference on Computer Systems (EuroSys)*. Nuremberg, Germany, 2009. 205, 208, 209, 213, 252

- [198] Ramanathan Narayanan, Berkin Ozisikyilmaz, Joseph Zambreno, Gokhan Memik, and Alok N. Choudhary. Minebench: A benchmark suite for data mining workloads. In *Proceedings of the 9th IEEE International Symposium on Workload Characterization (IISWC)*. San Jose, California, 2006. 45, 84, 95, 110, 126
- [199] Ripal Nathuji, Canturk Isci, and Eugene Gorbato. Exploiting platform heterogeneity for power efficient data centers. In *Proceedings of ICAC*. Jacksonville, FL, 2007. 13, 25, 65, 66, 196
- [200] Ripal Nathuji, Aman Kansal, and Alireza Ghaffarkhah. Q-clouds: Managing performance interference effects for qos-aware clouds. In *Proceedings of EuroSys*. Paris, France, 2010. 13, 18, 20, 26, 28, 57, 65, 135, 196
- [201] Intel nehalem architecture optimization reference manual. April 2012. 115
- [202] Aws case study: Netflix. <http://aws.amazon.com/solutions/case-studies/netflix/>. 10
- [203] Hiep Nguyen, Zhiming Shen, Xiaohui Gu, Sethuraman Subbiah, and John Wilkes. Agile: Elastic distributed resource scaling for infrastructure-as-a-service. In *Proceedings of ICAC*. San Jose, CA, 2013. 65, 81, 196
- [204] Dejan Novakovic, Nedeljko Vasic, Stanko Novakovic, Dejan Kostic, and Ricardo Bianchini. Deepdive: Transparently identifying and managing performance interference in virtualized environments. In *Proceedings of ATC*. San Jose, CA, 2013. 18, 66, 82, 139, 151, 196
- [205] Apache nutch. <http://nutch.apache.org/>. 98
- [206] Openstack cloud software. <http://www.openstack.org/>. 165
- [207] Simon Ostermann, Alexandru Iosup, Nezih Yigitbasi, Radu Prodan, Thomas Fahringer, and Dick Epema. A performance analysis of ec2 cloud computing services for scientific computing. In *Lecture Notes on Cloud Computing*. Volume 34, p.115-131, 2010. 166

- [208] Zhonghong Ou, Hao Zhuang, Jukka K. Nurminen, Antti Ylä-Jääski, and Pan Hui. Exploiting hardware heterogeneity within the same instance type of amazon ec2. In *Proceedings of HotCloud*. Boston, MA, 2012. 166
- [209] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. Sparrow: Distributed, low latency scheduling. In *Proceedings of SOSR*. Farmington, PA, 2013. 19, 20, 135, 136, 137, 138, 139, 145, 148, 156, 157, 182
- [210] Rina Panigrahy, Kunal Talwar, Lincoln Uyeda, and Udi Wieder. Heuristics for vector bin packing. In *Technical Report, 2011*. 250, 253
- [211] Abhay Parekh and Robert Gallager. A generalized processor sharing approach to flow control in integrated services networks: The single-node case. In *IEEE Transactions on Networks, Vol. 1, No. 3*. June 1993. 131
- [212] Gahyun Park. A generalization of multiple choice balls-into-bins. In *Proceedings of the 30th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*. San Jose, CA, 2011. 20, 138, 145
- [213] Nathan Parrish, Hyrum Anderson, Maya Gupta, and Dun Yu Hsaio. Classifying with confidence from incomplete information. In *Proceedings of the Journal Machine Learning Research (JMLR)*. 2013. 74, 75
- [214] Steven Pelley, David Meisner, Pooya Zandevakili, Thomas F. Wenisch, and Jack Underwood. Power routing: Dynamic power provisioning in the data center. In *Proceedings of the Fifteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Pittsburgh, Pennsylvania, USA, 2010. 253
- [215] Eduardo Pinheiro and Ricardo Bianchini. Energy conservation techniques for disk array-based servers. In *Proceedings of the 18th Annual International Conference on Supercomputing (ICS)*. Malo, France, 2004. 253
- [216] Nicolas Poggi, David Carrera, Ricard Gavaldà, Jordi Torres, and Eduard Ayguade. Characterization of workload and resource consumption for an online

- travel and booking site. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC'10)*, pages 1–10, 2010. 9
- [217] Presidents Council of Advisors on Science and Technology (PCAST). Designing a digital future: Federally funded research and development networking and information technology. <http://www.whitehouse.gov/sites/default/files/microsites/ostp/pcast-nitrd-report-2010.pdf>, December 2010. 9
- [218] Rackspace open cloud. <http://www.rackspace.com/>. 9, 24
- [219] Anand Rajaraman and Jeffrey Ullman. *Textbook on Mining of Massive Datasets*. 2011. 28, 45, 70, 74, 110, 115, 139, 147
- [220] Vijay Reddi, Benjamin Lee, Trishul Chilimbi, and Kushagra Vaid. Web search using mobile cores: Quantifying and mitigating the price of efficiency. In *Proc. of the International Symposium on Computer Architecture (ISCA)*. Saint-Malo, France, 2010. 9
- [221] Suhail Rehman and Majd Sakr. Initial findings for provisioning variation in cloud computing. In *Proceedings of CloudCom*. Indianapolis, IN, 2010. 166
- [222] Charles Reiss, Wilkes John, and Hellerstein John. Google cluster-usage traces: format + schema. In *Technical Report, Google Inc.* Mountain View, CA, USA, November 2011. 9
- [223] Charles Reiss, Alexey Tumanov, Gregory Ganger, Randy Katz, and Michael Kozych. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proceedings of SOCC*. 2012. 9, 60, 61, 66, 168, 174, 177
- [224] Rightscale. <https://aws.amazon.com/solution-providers/isv/rightscale>. 16, 57, 65, 176, 196
- [225] Alma Riska, James Larkby-Lahet, and Erik Riedel. Evaluating block-level optimization through the io path. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. Santa Clara, CA, 2007. 252

- [226] Roundcube. open-source webmail softwar. <http://roundcube.net/>. 98
- [227] Mohsen Salehi, Bahman Javadi, and Rajkumar Buyya. Preemption-aware admission control in a virtualized grid federation. In *Proceedings of the international conference on Advanced Information Networking and Applications (AINA)*. Fukuoka, Japan, 2012. 131
- [228] Daniel Sanchez and Christos Kozyrakis. Vantage: Scalable and Efficient Fine-Grain Cache Partitioning. In *Proceedings of the 38th annual International Symposium in Computer Architecture (ISCA-38)*. San Jose, CA, June, 2011. 45, 84, 110, 116, 126
- [229] Sriram Sankar and Kushagra Vaid. Addressing the stranded power problem in datacenters using storage workload characterization. In *Proceedings of WOSP/SIPEW*. San Jose, CA, 2010. 201, 209
- [230] Sriram Sankar and Kushagra Vaid. Storage characterization for unstructured data in online services applications. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*. Austin, TX, 2009. 201, 202, 205, 209
- [231] Jörg Schad, Jens Dittrich, and Jorge-Arnulfo Quiané-Ruiz. Runtime measurements in the cloud: Observing, analyzing, and reducing variance. *Proceedings VLDB Endow.*, 3(1-2):460–471, September 2010. 166
- [232] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. Omega: flexible, scalable schedulers for large compute clusters. In *Proceedings of EuroSys*. Prague, Czech Republic, 2013. 13, 19, 62, 63, 65, 78, 119, 135, 137, 138, 140, 149, 196
- [233] Malte Schwarzkopf, Derek G. Murray, and Steven Hand. The seven deadly sins of cloud computing research. In *Proceedings of HotCloud*, pages 1–1, 2012. 10
- [234] Shubhashis Sengupta and Rajeshwari Ganesan. Workload modeling for web-based systems. In *Proceedings of the 29th International Computer Measurement Group Conference*. Dallas, TX, 2003. 98, 259, 260

- [235] Jay Sethuraman and Mark Squillante. Optimal stochastic scheduling in multi-class parallel queues. In *Proceedings of SIGMETRICS*. Atlanta, Georgia, 1999. 133
- [236] Muhammad Zubair Shafiq, Lusheng Ji, Alex X. Liu, Jeffrey Pang, and Jia Wang. A first look at cellular machine-to-machine traffic: Large scale measurement and characterization. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*. London, England, UK, 2012. 258
- [237] Bikash Sharma, Victor Chudnovsky, Joseph L. Hellerstein, Rasekh Rifaat, and Chita R. Das. Modeling and synthesizing task placement constraints in google compute clusters. In *Proceedings of the 2nd ACM Symposium on Cloud Computing (SOCC)*. Cascais, Portugal, 2011. 145
- [238] Upendra Sharma, Prashant Shenoy, Sambit Sahu, and Anees Shaikh. A cost-aware elasticity provisioning system for the cloud. In *Proceedings of the 31st International Conference on Distributed Computing Systems (ICDCS)*. Minneapolis, MN, 2011. 16, 82
- [239] Daniel Shelepov, Juan Carlos Saez Alcaide, Stacey Jeffery, Alexandra Fedorova, Nestor Perez, Zhi Feng Huang, Sergey Blagodurov, and Viren Kumar. Hass: A scheduler for heterogeneous multicore systems. *OSP, vol. 43, 2009*. 18, 58, 115
- [240] Zhiming Shen, Sethuraman Subbiah, Xiaohui Gu, and John Wilkes. Cloudscale: elastic resource scaling for multi-tenant cloud systems. In *Proceedings of SOCC*. Cascais, Portugal, 2011. 17, 65, 196
- [241] David Shue, Michael J. Freedman, and Anees Shaikh. Performance isolation and fairness for multi-tenant cloud storage. In *Proc. of the 10th USENIX Conference on Operating Systems Design and Implementation, OSDI'12*, pages 349–362, Berkeley, CA, USA, 2012. USENIX Association. 20
- [242] Tajana Simunic and Steven Boyd. Managing power consumption in networks on chips. In *Proceedings of DAC*. Paris, France, 2002. 173

- [243] Aameek Singh, Madhukar Korupolu, and Dushmanta Mohapatra. Server-storage virtualization: Integration and load balancing in data centers. In *Proceedings of the ACM/IEEE Conference on Supercomputing (SC)*. Austin, Texas, 2008. 225, 253
- [244] Amy Spellmann, Karen Erickson, and Jim Reynolds. Server consolidation using performance modeling. In *IT Professional, 2003*. 252, 253
- [245] Sqlio disk subsystem benchmark tool. <http://www.microsoft.com/downloads/en/details.aspx?familyid=9a8b005b-84e4-4f24-8d65-cb53442d9e19&displaylang=en>. 204
- [246] Shekhar Srikantaiah, Aman Kansal, and Feng Zhao. Energy aware consolidation for cloud computing. In *Proceedings of the Conference on Power Aware Computing and Systems (HotPower)*. San Diego, CA, 2008. 253
- [247] Raymond T. Stefani, Bahram Shahian, Clement J. Savant, and Gene H. Hostetter. *Design of Feedback Control Systems*. 244
- [248] Christopher Stewart, Terence Kelly, and Alex Zhang. Exploiting non-stationarity for performance prediction. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems*. Lisbon, Portugal, 2007. 16
- [249] Mark Stillwell, David Schanzenbach, Frédéric Vivien, and Henri Casanova. Resource allocation algorithms for virtualized service hosting platforms. *J. Parallel Distrib. Comput.*, 70(9):962–974, September 2010. 17, 250, 253
- [250] Mark Stillwell, Frédéric Vivien, and Henri Casanova. Virtual machine resource allocation for service hosting on heterogeneous distributed platforms. In *Proc. of the 26th International Parallel and Distributed Processing Symposium (IPDPS)*. Shanghai, 2012. 17
- [251] Alexander Stolyar. On the stability of multiclass queueing networks: A relaxed sufficient condition via limiting fluid processes. In *Technical Report, 1995*. 132

- [252] Storm. <https://github.com/nathanmarz/storm/>. 84, 160
- [253] Jimeng Sun, Yinglian Xie, Hui Zhang, and Christos Faloutsos. Less is more: Compact matrix decomposition for large sparse graphs. In *Proceedings of SDM*. Minneapolis, MN, 2007. 30
- [254] Steven Swanson, Andrew Putnam, Martha Mercaldi, Ken Michelson, Andrew Petersen, Andrew Schwerin, Mark Oskin, and Susan J. Eggers. Area-performance trade-offs in tiled dataflow architectures. In *Proceedings of ACM SIGARCH Computer Architecture News, v.34 n.2, p.314-326, May 2006*. 168
- [255] Lingjia Tang, Jason Mars, and Mary-Lou Soffa. Compiling for niceness: Mitigating contention for qos in warehouse scale computers. In *Proceedings of CGO*. San Jose, CA, 2012. 98, 99
- [256] Wenting Tang, Yun Fu, Ludmila Cherkasova, and Amin Vahdat. Medisyn: A synthetic streaming media service workload generator. In *Proceedings of the 13th International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV)*. Monterey, CA, USA, 2003. 259
- [257] Kanit Therdsteeerasukdi, Gyungsu Byun, Jason Cong, Frank Chang, and Glenn Reinman. Utilizing rf-i and intelligent scheduling for better throughput/watt in a mobile gpu memory system. In *TACO 8(4)*. 2012. 168
- [258] Eno Thereska, Austin Donnelly, and Dushyanth Narayanan. Sierra: Practical power-proportionality for data center storage. In *Proceedings of the Sixth Conference on Computer Systems (EuroSys)*. Salzburg, Austria, 2011. 226, 227, 238, 239, 244, 251, 253
- [259] Torque resource manager. <http://www.adaptivecomputing.com/products/open-source/torque/>. 13, 65, 196
- [260] Tpc benchmarks. <http://www.tpc.org/tpcc/>. 212
- [261] Bhuvan Urgaonkar, Prashant Shenoy, and Timothy Roscoe. Resource overbooking and application profiling in shared hosting platforms. In *Proceedings of*

- the 5th Symposium on Operating Systems Design and Implementation (OSDI)*. Boston, MA, 2002. 16
- [262] K Vaid. Datacenter power efficiency: Separating fact from fiction. Invited talk at the 2010 Workshop on Power Aware Computing and Systems, October 2010. 10, 11
- [263] Kenzo Van Craeynest, Aamer Jaleel, Lieven Eeckhout, Paolo Narvaez, and Joel Emer. Scheduling heterogeneous multi-cores through performance impact estimation (pie). In *Proceedings of the International Symposium on Computer Architecture (ISCA), Portland, OR, 2012*. 18, 58, 115, 116
- [264] Henk Vandenbergh. Vdbench: User guide 5.00. 204
- [265] Arunchandar Vasam, Anand Sivasubramaniam, Vikrant Shimpi, T. Sivabalan, and Rajesh Subbiah. Worth their watts? an empirical study of datacenter servers. In *Proceedings of the 16th International Symposium on High Performance Computer Architecture (HPCA)*. Bangalore, India, 2010. 61, 66
- [266] Nedeljko Vasić, Dejan Novaković, Svetozar Miućin, Dejan Kostić, and Ricardo Bianchini. Dejavu: accelerating resource allocation in virtualized environments. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. London, UK, 2012. 17, 26, 28, 57, 65, 196
- [267] 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 *Proceedings of the Symposium on Cloud Computing*. Santa Clara, CA, 2013. 13, 19, 135
- [268] VMware vcloud suite. <http://www.vmware.com/products/vcloud>. 165
- [269] Virtualbox. <https://www.virtualbox.org/>. 82, 151

- [270] VMware virtual machines. <http://www.vmware.com/>. 62, 82, 151
- [271] VMWare-DRS. Distributed resource scheduler: Design, implementation and lessons learned. In *VMware Technical Journal*, 1(1), 2012. 18, 110
- [272] Automated reclamation part: Recovering inactive resources. <http://blogs.vmware.com/management/2014/02/automated-reclamation-part-3-recovering-inactive-resources.html>. 17
- [273] VMware vsphere. <http://www.vmware.com/products/vsphere/>. 18, 24, 42, 57, 227, 251
- [274] Carl Waldspurger. Memory resource management in vmware esx server. In *Proc. of the Fifth Symposium on Operating Systems Design and Implementation (OSDI)*. Boston, MA, 2002. 17
- [275] Guohui Wang and T. S. Eugene Ng. The impact of virtualization on network performance of amazon ec2 data center. In *Proceedings of INFOCOM*. San Diego, CA, 2010. 74, 166
- [276] Brian J. Watson, Manish Marwah, Daniel Gmach, Yuan Chen, Martin Arlitt, and Zhikui Wang. Probabilistic performance modeling of virtualized resource allocation. In *Proceedings of the 7th International Conference on Autonomic Computing*. Washington, DC, 2010. 16
- [277] Thomas Wensich, Roland Wunderlich, Michael Ferdman, Anastassia Ailamaki, Babak Falsafi, and James Hoe. Simflex: Statistical sampling of computer system simulation. *IEEE MICRO*, vol. 26, no. 4, Jul./Aug. 2006. 32
- [278] Jonathan Wildstrom, Peter Stone, Emmett Witchel, Raymond J. Mooney, and Michael Dahlin. Towards self-configuring hardware for distributed computer systems. In *Proceedings of the Second International Conference on Autonomic Computing (ICAC) 2005*. Seattle, WA, 2005. 22
- [279] Windows azure. <http://www.windowsazure.com/>. 9, 18, 24, 44, 57, 165, 225

- [280] Jonathan Winter and David Albonesi. Scheduling algorithms for unpredictably heterogeneous cmp architectures. In *Proceedings of DSN*. 2008. 115
- [281] Ian H. Witten, Eibe Frank, and Geoffrey Holmes. *Data Mining: Practical Machine Learning Tools and Techniques*. 3rd Edition. 30, 70, 74, 139
- [282] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The splash-2 programs: characterization and methodological considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture (ISCA)*. Santa Margherita Ligure, Italy, 1995. 45, 84, 95, 110, 126
- [283] The xen project. <http://www.xen.org/>. 26, 82, 151
- [284] Zhen Xiao, Weijia Song, and Qi Chen. Dynamic resource allocation using virtual machines for cloud computing environment. In *IEEE Transactions on Parallel and Distributed Systems, Vol. 24, Issue: 6*. September 2012. 17
- [285] Hailong Yang, Alex Breslow, Jason Mars, and Lingjia Tang. Bubble-flux: precise online qos management for increased utilization in warehouse scale computers. In *Proceedings of ISCA*. 2013. 17, 20, 62, 65, 66, 135, 139, 196
- [286] Benjamin Yolken and Nick Bambos. Game based capacity allocation for utility computing environments. In *Proceedings of Gamecomm*. Athens, Greece, 2008. 133
- [287] Minlan Yu, Albert Greenberg, Dave Maltz, Jennifer Rexford, Lihua Yuan, Srikanth Kandula, and Changhoon Kim. Profiling network performance for multi-tier data center applications. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation (NSDI)*. Boston, MA, 2011. 257
- [288] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient

- distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of NSDI*. San Jose, CA, 2012. 10, 20, 84, 135, 136, 166
- [289] Matei Zaharia, Andy Konwinski, Anthony Joseph, Randy Katz, and Ion Stoica. Improving mapreduce performance in heterogeneous environments. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. San Diego, CA, 2008. 72, 82
- [290] Seyed Majid Zahed and Benjamin C. Lee. Ref: Resource elasticity fairness with sharing incentives for multiprocessors. In *Proceedings of ASPLOS*. Salt Lake City, UT, 2014. 21, 197
- [291] Zfs. <http://www.freebsd.org/doc/handbook/filesystems-zfs.html>. 82
- [292] Alex Zhang, Fereydoon Safai, and Dirk Beyer. Server consolidation: High-dimensional probabilistic bin-packing. In *Proceedings of Informs*. San Francisco, CA, 2005. 253
- [293] Jianyong Zhang, Anand Sivasubramaniam, Hubertus Franke, Natarajan Gautam, Yanyong Zhang, and Shailabh Nagar. Synthesizing representative i/o workloads for tpc-h. In *Proceedings of the 10th International Symposium on High Performance Computer Architecture (HPCA)*. Madrid, Spain, 2004. 157
- [294] Joy Ying Zhang, Pang Wu, Jiang Zhu, Hao Hu, and Flavio Bonomi. Privacy-preserved mobile sensing through hybrid cloud trust framework. In *Proceedings of ICCS*. Shenzhen, China, 2013. 21, 197
- [295] Qi Zhang, Mohamed Faten Zhani, Shuo Zhang, Quanyan Zhu, Raouf Boutaba, and Joseph L. Hellerstein. Dynamic energy-aware capacity provisioning for cloud computing environments. In *Proceedings of the 9th international conference on Autonomic computing*, pages 145–154, 2012. 9
- [296] Xiao Zhang, Eric Tune, Robert Hagmann, Rohit Jnagal, Vrigo Gokhale, and John Wilkes. Cpi2: Cpu performance isolation for shared compute clusters. In *Proceedings of EuroSys*. Prague, Czech Republic, 2013. 17, 20, 66, 96, 135, 196

- [297] Li Zhao, Ravi Iyer, Jaideep Moses, Ramesh Illikkal, Srihari Makineni, and Don Newell. Exploring large-scale cmp architectures using manysim. In *IEEE Micro*, vol.27 n.4, July 2007. 168
- [298] Wei Zheng, Ricardo Bianchini, G. John Janakiraman, Jose Renato Santos, and Yoshio Turner. Justrunit: experiment-based management of virtualized data centers. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. San Diego, CA, 2009. 16
- [299] Zhang Zhi-Hong, Meng Dan, Zhan Jian-Feng, Wang Lei, Wu Lin-ping, and Huang We. Easy and reliable cluster management: The self-management experience of fire phoenix. In *Proc. of the 20th International Parallel and Distributed Processing Symposium (IPDPS)*. Rhodes Island, 2006. 13
- [300] Xiaoyun Zhu, Donald Young, Brian J. Watson, Zhikui Wang, Jerry Rolia, Sharad Singhal, Bret Mckee, Chris Hyser, Daniel Gmach, Robert Gardner, Tom Christian, and Ludmila Cherkasova. 1000 islands: An integrated approach to resource management for virtualized datacenters. *Journal of Cluster Computing*, vol. 12, 2009. 16, 17, 57
- [301] Tsahee Zidenberg, Isaac Keslassy, and Uri Weiser. Multiamdahl: How should i divide my heterogenous chip? In *Computer Architecture Letters (CAL)*, vol. 11. 2012. 115