Dynamic Server Provisioning for Data Center Power Management

Anshul Gandhi

CMU-CS-13-110

June 2013

Computer Science Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Mor Harchol-Balter, Chair
David G. Andersen
Jeffrey O. Kephart, IBM Research
Alan Scheller-Wolf
Karsten Schwan, GeorgiaTech

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

Copyright © 2013 Anshul Gandhi

This research was sponsored by the National Science Foundation under grant numbers CCR-0615262 and CSR-116282; by an IBM Faculty Award; by an MSR/CMU computational thinking grant; and by a grant from the Intel Science and Technology Center on Cloud Computing

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

Keywords: Capacity Management, Data Centers, Dynamic Server Provisioning, Performance Modeling, Power Management, Queueing Theory, Resource Provisioning, Setup Costs

For my family and friends, for keeping me in touch with the finer things in life.

Abstract

Data centers play an important role in today's IT infrastructure. However, their enormous power consumption makes them very expensive to operate. Sadly, much of the power used by data centers is wasted because of poor capacity management, leading to low server utilization.

In order to reduce data center power consumption, researchers have proposed several dynamic server provisioning approaches. However, there are many challenges that hinder the successful deployment of dynamic server provisioning, including: (i) unpredictability in workload demand, (ii) switching costs when setting up new servers, and (iii) unavailability of data when provisioning stateful servers. Most of the existing research in dynamic server provisioning has ignored, or carefully sidestepped, these important challenges at the expense of reduced benefits. In order to realize the full potential of dynamic server provisioning, we must overcome these associated challenges.

This thesis provides new research contributions that explicitly address the open challenges in dynamic server provisioning. We first develop novel performance modeling tools to estimate the effect of these challenges on response time and power. In doing so, we also address several long-standing open questions in queueing theory, such as the analysis of multi-server systems with switching costs. We then present practical dynamic provisioning solutions for multi-tier data centers, including novel solutions that allow scaling the stateful caching tier and solutions that are robust to load spikes. Our implementation results using realistic workloads and request traces on a 38-server multi-tier testbed demonstrate that dynamic server provisioning can successfully meet typical response time guarantees while significantly lowering power consumption.

While this thesis focuses on server provisioning for reducing power in data centers, the ideas presented herein can also be applied to: (i) private clouds, where unneeded servers can be repurposed for "valley-filling" via batch jobs, to increase server utilization, (ii) community clouds, where unneeded servers can be given away to other groups, to increase the total throughput, and (iii) public clouds, where unneeded virtual machines can be released back to the cloud, to reduce rental costs.

Acknowledgments

This thesis has been made possible by the help of the many people who helped me throughout my PhD. First and foremost, I would like to thank my advisor, Mor Harchol-Balter, for her constant support and encouragement, and for all the time and effort she has devoted towards my research. I have learnt a lot from Mor during my PhD, including how to approach research problems, how to write technical papers, and how to give good talks. I am also grateful to my thesis committee members, Dave Andersen, Jeff Kephart, Alan Scheller-Wolf, and Karsten Schwan, for helping me shape this thesis. Alan has additionally been a second "theory" advisor for me these past six years, and I have always enjoyed working with him on the numerous theory problems that we tried to tackle. Mike Kozuch has been my second "systems" advisor, and probably the most fun person I have ever worked with. I would also like to thank my past and present colleagues, Varun Gupta, Timothy Zhu, and Sherwin Doroudi, for the many interesting and insightful discussions. A big thank you to Deb Cavlovich, Nicole Stenger, Carly Shane, Nancy Conway, and Charlotte Yano for their invaluable administrative support during my stay at CMU.

I have been fortunate to have had great friends during my PhD. Amith Darbal, Danish Faruqui, Dhishan Kande, Debasis Kar, Satyajeet Ojha, Neerav Verma, you have seen first-hand how much effort actually went into making this thesis possible. Thanks for all your help, and for making my PhD so much fun!

I have also been very fortunate to have had the support of my best friend and wife, Sneha, throughout my PhD. Sneha's unconditional love gave me the strength to successfully complete this thesis. I must also mention that Sneha's perseverance in her medical career made it very difficult for me to get lazy. Thanks for keeping me going!

Finally, I would like to thank my parents and my sister for their endless love and support, and for trusting my decision to pursue a PhD. You guys will always be my greatest advisors!

Contents

1 Introduction **1**

1.1 Data Centers . . . 2

1.1.1 Performance metrics . . . 3

1.2 Need for Power Management in Data Centers . . . 4

1.2.1 Where does the power come from? . . . 4

1.2.2 Where does the power go in a data center? . . . 4

1.2.3 Power metrics . . . 5

1.3 Data Center Power Management Approaches . . . 5

1.3.1 Power-proportionality . . . 6

1.3.2 Energy-efficient Server Design . . . 7

1.3.3 Dynamic Server Provisioning . . . 8

1.3.4 Consolidation and Virtualization . . . 11

2 Thesis Overview **13**

2.1 Dynamic Server Provisioning . . . 13

2.1.1 Motivation behind dynamic server provisioning . . . 14

2.1.2 Why is dynamic server provisioning difficult? . . . 15

2.1.3 The need for this thesis . . . 16

2.2 Prior Work in Dynamic Server Provisioning . . . 17

2.2.1 Predictive approaches . . . 17

2.2.2 Reactive approaches . . . 18

2.2.3 Hybrid approaches . . . 19
2.2.4 Dynamic provisioning for stateful servers . . . 19
2.2.5 Decentralized dynamic server provisioning approaches . . . 20
2.2.6 Dynamic provisioning approaches in operations research . . . 21
2.3 Scope of this Thesis . . . 22
2.3.1 Metrics . . . 22
2.3.2 Implementation testbed . . . 24
2.3.3 Research questions addressed by this thesis . . . 26
2.4 Novelty and Contributions . . . 27

3 Challenges in Dynamic Server Provisioning 31
3.1 Setup Costs . . . 32
3.1.1 Analysis of setup costs . . . 32
3.1.2 Effect of scale on setup costs . . . 38
3.2 Uncertainty in Workload Demand . . . 40
3.2.1 Analysis of workload traces . . . 40
3.2.2 Load spikes . . . 43
3.3 Stateful Servers . . . 46
3.3.1 Consequences of dynamically changing the cache size . . . 46
3.4 Chapter Summary . . . 48

4 AutoScale: Robust Dynamic Server Provisioning 49
4.1 Introduction . . . 50
4.2 Experimental Setup . . . 53
4.2.1 Our experimental testbed . . . 53
4.2.2 Trace-based arrivals . . . 54
4.3 Evaluation I: Fluctuations in Request Rate . . . 54
4.3.1 AlwaysOn . . . 55
4.3.2 Reactive . . . 56

4.3.3 Reactive with extra capacity 57
4.3.4 Predictive 57
4.3.5 AutoScale−− 58
4.3.6 Opt 61
4.4 Wear-and-tear costs of server provisioning 62
4.5 Impact of Sleep States 63
4.5.1 Lower setup times 64
4.5.2 Lower sleep power 66
4.5.3 Sensitivity analysis of sleep states 67
4.6 Impact of Lower Idle Power 69
4.7 Evaluation II: Robustness 71
4.7.1 Why request rate is not a good feedback signal 71
4.7.2 A better feedback signal that is still not quite right 72
4.7.3 AutoScale: Incorporating the right feedback signal 74
4.7.4 Alternative feedback signal choices 80
4.8 Prior Work 81
4.9 Chapter Summary 82

5 SoftScale: A Novel Approach to Handling Load Spikes 85

5.1 Introduction 86
5.2 Our Experimental Testbed 90
5.2.1 Workload 90
5.2.2 Provisioning 91
5.3 SoftScale 92
5.3.1 When to invoke SoftScale? 93
5.3.2 How much application work can memcached handle? 94
5.3.3 Need for isolation 95
5.3.4 The SoftScale algorithm 96
5.3.5 Analytical model for estimating *SoftScale*'s performance 97

5.4 Results 97
5.4.1 Characterizing the range of load jumps that *SoftScale* can handle 98
5.4.2 Spikes in real-world traces 101
5.4.3 Spikes created by server faults 102
5.5 Lower Setup Times 103
5.6 Future Architectures 105
5.7 Prior Work 106
5.8 Chapter Summary 108

6 CacheScale: Dynamic Provisioning of the Caching Tier 111
6.1 Introduction 112
6.2 Experimental Setup 114
6.3 Assessment of Cache Savings 115
6.3.1 Popularity distribution 115
6.3.2 Theoretical model 116
6.3.3 Theoretical results 117
6.3.4 Experimental results 118
6.4 CacheScale 119
6.4.1 Scaling down 119
6.4.2 Scaling up 120
6.4.3 Alternative approaches 121
6.5 Chapter Summary 121

7 Recursive Renewal Reward 123
7.1 Introduction 124
7.2 Prior Work 125
7.2.1 Matrix-analytic based approaches 125
7.2.2 Generating function based approaches 126
7.2.3 M/M/k with vacations 126

7.2.4 Restricted models of M/M/k with setup 126
7.2.5 How our work differs from all of the above 127
7.3 Model 127
7.4 The Recursive Renewal Reward technique 128
7.5 M/M/1/setup 129
7.5.1 Deriving $\mathcal{T}$ via $\mathbf{T^L_{0,1}}$ and $\mathbf{T^L_{1,1}}$ 130
7.5.2 Deriving $\mathcal{R}$ via $\mathbf{R^L_{0,1}}$ and $\mathbf{R^L_{1,1}}$ 132
7.5.3 Deriving $\mathbf{E}[\mathbf{N}]$ 133
7.5.4 Deriving $\widehat{\mathbf{N}}(\mathbf{z})$ and $\widetilde{\mathbf{T}}(\mathbf{s})$ 134
7.5.5 Deriving $\widehat{\mathbf{P}}(\mathbf{z})$ 135
7.6 M/M/k/setup 136
7.6.1 System of equations for $\mathbf{p^L_{i \to d}}$ 137
7.6.2 Deriving $\dot{\mathbf{R}}^{\mathbf{L}}_{\mathbf{i,k}}$ for the repeating portion 138
7.6.3 Deriving $\dot{\mathbf{R}}^{\mathbf{H}}_{\mathbf{i,j}}$ for the non-repeating portion 138
7.7 The Generalized Recursive Renewal Reward technique 139
7.8 Chapter Summary 142

8 Related Work **143**
8.1 Power-proportionality 143
8.1.1 Lower server idle power 143
8.1.2 Voltage and frequency scaling 144
8.2 Energy-efficient Server Design 145
8.2.1 Low-power inactive states 145
8.2.2 Heterogenous designs 145
8.2.3 Energy-efficient cluster architectures 145
8.3 Consolidation and Virtualization 146
8.4 Other Approaches 147

9 Conclusion **149**

9.1 Contributions 149
9.1.1 Analyzing the challenges in dynamic provisioning 149
9.1.2 AutoScale: Overcoming the challenges presented by setup costs and unpredictable workload demand 150
9.1.3 SoftScale: A new approach to handling load spikes 150
9.1.4 CacheScale: Dynamically scaling the caching tier 151
9.1.5 Recursive Renewal Reward: A new technique for solving Markov chains with a repeating structure 151
9.2 Future Work 152

Bibliography **155**

List of Figures

1.1 Abstract data center model 2

2.1 Our implementation testbed 24

3.1 Our M/M/k queueing model for a multi-server system 32
3.2 M/M/k/setup Markov chain 34
3.3 Effect of setup costs for t_{setup} = 1s and mean request size of 1s 36
3.4 Effect of setup costs for t_{setup} = 10s and mean request size of 1s . . . 36
3.5 Effect of setup costs for t_{setup} = 100s and mean request size of 1s . . . 36
3.6 Effect of setup costs for t_{setup} = 100s and mean request size of 100s . 37
3.7 Effect of scale on setup costs for mean request size of 1s 38
3.8 Effect of scale on setup costs for mean request size of 100s 38
3.9 Effect of scale on setup costs for mean request size of 1s under deterministic setup times 39
3.10 Effect of scale on setup costs for mean request size of 100s under deterministic setup times 39
3.11 Demand plots for a single day for our traces 41
3.12 Time-series demand plots for our traces 42
3.13 Consequences of load spikes 45
3.14 Load spikes under various setup times 45
3.15 Consequences of dynamically changing the cache size 47

4.1 Traces used for experimental evaluation in this chapter 54

4.2 How many req/s can a single server handle? 55
4.3 *AlwaysOn* 55
4.4 *Reactive* 56
4.5 *Predictive* 58
4.6 Packing factor for a single server 60
4.7 *AutoScale--* 60
4.8 *Opt* 61
4.9 Effect of lower setup times 65
4.10 Effect of lower sleep power 66
4.11 Sensitivity analysis of sleep states for the Big spike trace 68
4.12 Sensitivity analysis of sleep states for the Dual phase trace 68
4.13 Effect of lower idle power 70
4.14 Consequences of changing the request size 72
4.15 New feedback signal for provisioning: number of requests in the system 72
4.16 Number of requests in system overestimates required capacity 73
4.17 Non-linear relationship between request rate and number of requests . 73
4.18 Number of requests is a robust feedback signal for changes in load . . 75
4.19 Robustness of *AutoScale* to changes in request size 79

5.1 A preview of *SoftScale*'s abilities 87
5.2 Our experimental testbed for *SoftScale* 90
5.3 Provisioning the application tier 91
5.4 Design decisions for *SoftScale* 94
5.5 Enhancing *SoftScale* using dynamic core isolation 96
5.6 Illustration of load jumps 96
5.7 *SoftScale* successfully handles a range of load jumps without consuming extra resources 98
5.8 Full range of results for *SoftScale* 99
5.9 Real-world trace snippets used for our experiments in this chapter . . 101

5.10 *SoftScale*'s superiority over the baseline for the Pi Day trace 102
5.11 *SoftScale* can even handle load spikes created by server failures 103
5.12 *SoftScale* under lower setup times for a 15% → 30% load jump 104
5.13 *SoftScale* under lower setup times for a 20% → 50% load jump 104
5.14 Full range of results for *SoftScale* under a 20s setup time 104
5.15 Processor architectures with more cores benefit *SoftScale* 106
5.16 Full range of results for *SoftScale* with 8-core memcached servers . . . 107

6.1 Illustration of a multi-tier cloud service 112
6.2 A small decrease in cache hit rate can lead to a large decrease in the amount of cached data 114
6.3 The Zipf popularity distribution 116
6.4 Theoretical cache savings for a given peak-to-min ratio 117
6.5 *CacheScale* dynamically provisions the caching tier without violating response time SLAs 120

7.1 M/M/k/setup Markov chain 128
7.2 M/M/1/setup Markov chain 130
7.3 Class of Markov chains that can be analyzed via RRR 140

List of Tables

4.1 The (in)sensitivity of *AutoScale*--'s performance to t_{wait} 59

4.2 Comparison of dynamic provisioning policies via implementation . . . 62

4.3 Description of variables used in *AutoScale*'s capacity inference algorithm 76

4.4 Comparison of dynamic server provisioning policies for 2x request size 78

4.5 Comparison of dynamic server provisioning policies for 4x request size 78

4.6 Comparison of dynamic server provisioning policies for lower CPU frequency 80

7.1 Variables used in our mean analysis 132

7.2 Variables used in our transform analyses 134

List of Definitions, Observations, and Theorems

1.1 Definition (Response Time) 3
1.2 Definition (Service Level Agreement) 3
1.3 Definition (Setup Time) 9

2.1 Definition (T_{avg}) 22
2.2 Definition (T_{95}) 23
2.3 Definition (Instantaneous T_{95}) 23
2.4 Definition (P_{avg}) 23
2.5 Definition (N_{avg}) 23
2.6 Definition (PPW) 23

3.1 Definition (Load) 33
3.1 Observation (The challenge presented by setup costs) 37
3.2 Observation (Factors affecting setup costs) 37
3.3 Observation (Effect of scale on setup costs) 39
3.4 Observation (Variance in workload demand) 41
3.5 Observation (Predictability of workload demand) 41
3.6 Observation (The challenge presented by short-term fluctuations) . . 43
3.7 Observation (The challenge presented by load spikes) 44
3.8 Observation (The challenge presented by stateful servers) 47

4.1 Observation (Dynamic server provisioning under lower setup times) . 65
4.2 Observation (Dynamic server provisioning under lower sleep power) . 66
4.3 Observation (Usefulness of sleep states) 69
4.4 Observation (Dynamic server provisioning under lower idle power) . . 70

7.1 Theorem (Recursion theorem for mean time) 131
7.2 Theorem (Recursion theorem for mean reward) 132
7.3 Theorem (Recursion theorem for transform of reward) 134
7.4 Theorem (Recursion theorem for transform of power) 135
7.5 Theorem (Recursion theorem for probability) 136

Chapter 1

Introduction

Data centers are an integral part of today's internet services. Business organizations and corporations around the world rely heavily on data centers for their daily operations. In fact, every time we search for a query on the internet, or use an application on our smartphones, we make use of data centers.

At a high level, a data center is simply a massive collection of servers that stores data and provides computational resources. A primary goal for data center operators is to ensure that users, or customers, receive good performance. This often translates to having sufficient capacity (number of servers) to provide the required performance. However, given the size of today's data centers, a secondary goal is to minimize the operational costs of running the data center. The power consumed by the servers is often the biggest contributing factor to the data center's operational expenses [26, 153]. Thus, an important question for data center operators is *how to manage capacity so as to optimize the power-performance tradeoff*.

In this chapter we give a broad overview of data centers (Section 1.1) and motivate the need for power management in data centers (Section 1.2). We then discuss the various approaches to data center power management, and the challenges associated with each of them (Section 1.3).

In the next chapter, we provide an overview of this thesis. In particular, Chapter 2 focuses on the scope of this thesis: dynamic server provisioning. Chapter 3 examines the challenges in dynamic server provisioning. We then provide practical solutions for these challenges in Chapters 4, 5, and 6. Chapter 7 presents our new theoretical work on analyzing dynamic server provisioning. We discuss related work in Chapter 8, and conclude with a summary of this thesis in Chapter 9.

1.1 Data Centers

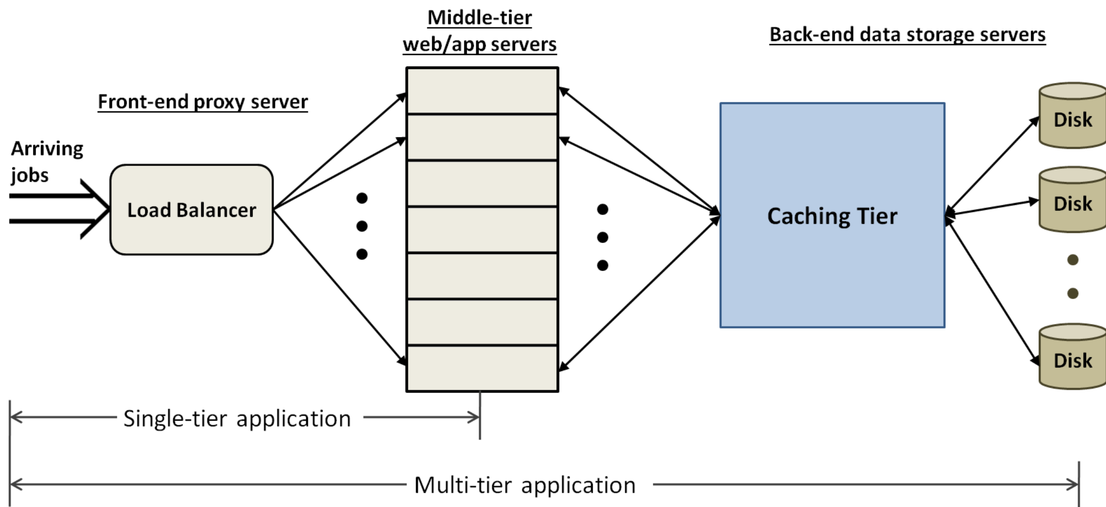


Figure 1.1: Abstract data center model.

A data center is a facility that physically houses a number of connected computer servers and provides the necessary infrastructure, such as the power, heating, ventilation and air conditioning system, to keep them operational. In this thesis, we will focus specifically on the servers within a data center.

Figure 1.1 illustrates the server layout in a typical data center architecture. Incoming requests first arrive at a load balancer, or a *front-end proxy server*, and are then distributed among *middle-tier web servers or application servers*. These servers are typically *stateless, meaning they do not store any data that is required for serving the requests*. The middle-tier servers parse the incoming request and determine the data needed, such as customer profiles or account information, to satisfy the request. This data is then fetched from the *back-end data storage servers*. These data servers are *stateful, meaning they store data that is necessary to serve customer requests*. Traditionally, the back-end servers store persistent data on hard drives/disks, usually in the form of databases. However, accessing data via the disk significantly increases the application response time. To improve performance, data centers today typically make use of a caching tier, which reduces data fetch latency. A popular cache management software used in caching tiers is memcached [59]. With memcached, most requests for dynamic content will be served by the caching tier servers, and only a fraction of the requests will have to incur the huge performance hit associated with accessing data via the hard disk. Popular multi-tier services include social networking (for example, Facebook), online banking, e-commerce companies (for example,

Amazon, e-bay), etc. However, not all applications are multi-tiered. For example, websites for small businesses, personal home pages, compute-intensive applications such as simulations, etc, are all single-tier applications. Such applications usually only require the middle-tier web/application servers. Other popular examples of single-tier architecture include call centers, service centers, manufacturing systems, etc.

Data centers host a variety of workloads ranging from multi-tier web workloads (online services such as Amazon [48], Google [33]) to data-intensive (MapReduce [47], Hadoop [165]) and compute-intensive (scientific computing, super-computing) workloads. Data centers are also used to host virtual machines (such as Amazon EC2 [13]), which can then be leased, or rented, to customers. The specific workload running on these virtual machines depends on the user.

1.1.1 Performance metrics

Performance is often the most important metric for data center operators. Since data centers host a range of workloads, performance is a very subjective term. For data hosting applications, performance is often measured in terms of availability of data over a period of time. For batch workloads, performance is often measured in terms of throughput or completion time. For web workloads or database queries, a popular performance metric is response time. In this thesis, we use response time as our performance metric. Formally, response time is defined as follows:

Definition 1.1 (Response Time)**.**
Response time for a request is defined as the time (in seconds or milliseconds) from when the request arrives at the data center to the time when it completes execution.

Data centers that serve customers often provide a guarantee on the response time experienced by a customer request. An agreement that reflects this guarantee is called a Service Level Agreement (SLA). Formally, an SLA for response time can be defined as follows:

Definition 1.2 (Service Level Agreement)**.**
A Service Level Agreement (SLA) is a contract between the service provider and the customer that states an upper bound on the response time a customer will experience for a given request, with a certain probability, under certain conditions.

An example of an SLA (from Amazon [48]) is a service guaranteeing its users a

response time below 300ms for 99.9% of the requests, given a peak load of no more than 500 requests per second.

1.2 Need for Power Management in Data Centers

Data centers often house thousands of servers [104] to provide good response times to their huge customer base. With the emergence of cloud computing and the growth in big data, the number of servers in a data center will rise even higher. At such scales, the power consumed by the servers becomes a significant operational expense.

Data center power consumption within the US accounts for more than 1.5% of the total electricity usage. The rising cost of energy and the tremendous growth of data centers will result in even more spending on power consumption. Based on EPA estimates, the energy consumption in US data centers exceeded 100 billion kWh in 2011, at a cost of $7.4 billion [11]. *Rising energy costs, regulatory requirements and social concerns over greenhouse gas emissions have made it critical to reduce power consumption in data centers.*

1.2.1 Where does the power come from?

Data centers typically satisfy their electricity needs by using grid power from a utility company. Of course, most data centers also have uninterrupted power supply (UPS) on site to deal with power outages. Some data centers also harness power from renewable energy sources to offset their grid power consumption [35, 137, 83]. Nevertheless, grid power remains the primary source for almost all data centers today.

1.2.2 Where does the power go in a data center?

Of the power supplied to a data center, roughly 50% is consumed by site infrastructure such as power delivery systems and cooling systems [11]. An additional 10% goes into powering the network equipment and the storage systems. The remaining 40% is consumed by the servers themselves. Thus, *for every $1 spent on powering a server, an additional $1.5 is spent on the supporting infrastructure.*

1.2.3 Power metrics

The two important power metrics that data center operators care about are (i) average power consumption, and (ii) peak power consumption.

The price that a data center operator pays, say monthly, for power consumption depends on the total energy used in that month, typically in units of kilowatt-hours (kWh). Thus, reducing the average power consumption directly results in cost savings. Note that reducing the power consumption of the servers also helps reduce the associated power required for cooling these servers.

Peak power consumption is important because data center infrastructure is often provisioned for a specific peak power budget [57]. For example, when provisioning the air conditioning system, the power delivery infrastructure, or the circuit breakers, a specific peak power consumption is assumed [106, 133]. If the peak power consumption of the data center exceeds this finite power budget, companies will have to invest in building additional data centers. Thus, there is a huge cost-savings incentive to limit peak power consumption in data centers.

The specific power metrics considered in this thesis will be discussed in Section 2.3.

1.3 Data Center Power Management Approaches

Traditionally, data center research has focussed on trying to maximize performance. However, given the significance of power usage in today's data centers, researchers are now focussing on the joint problem of maximizing performance while minimizing power consumption. In this section we list the popular approaches that have been employed for data center power management, and the important challenges encountered by each of the approaches. A detailed study of the prior work employing these approaches will be discussed later in Chapter 8.

There are four different high-level approaches for reducing power consumption in data centers:

1. Power-proportionality:
   Finding ways to ensure that servers that are on consume power in proportion to their utilization.
2. Energy-efficient Server Design:
   Building the right server architecture for a given workload.

3. Dynamic Server Provisioning:
   Turning servers on and off at the right times.
4. Consolidation and Virtualization:
   Amortizing power consumption by resource sharing.

While this thesis focuses on dynamic server provisioning, it is instructive to briefly discuss all the above approaches.

1.3.1 Power-proportionality

A popular approach to power management is to ensure that servers that are turned on consume only the bare minimum amount of power needed to service customer requests [26]. This requires the servers to be "power-proportional," meaning that servers consume power in proportion to the load they experience. The basic idea here is that servers should consume very little power when they are idle. When the servers do experience load, leading to a utilization of say $x\%$, then they should only consume $x\%$ of their peak power, and no more. Power-proportionality has motivated a lot of research and development in recent years. Most of the focus, however, has been on reducing system idle power consumption. Low idle power has motivated the novel workload scheduling idea of "Race to Idle," which advocates executing existing workload continuously at maximum speed, and then transitioning to the idle mode until a certain amount of new workload has accumulated. There has been some progress on power-proportionality for cases when there is load in the system. Dynamic Voltage and Frequency Scaling (DVFS) allows the processor to consume less power (when compared to peak power) by running at a lower voltage and frequency setting. DVFS is a very practical solution because it takes less than a millisecond [43] to transition between different Voltage and Frequency settings [87]. We now list the research challenges associated with power-proportionality:

- How to reduce a server's idle power consumption?
  Processor designers have been very successful at reducing the CPU idle power [89, 109]. However, other server components still consume significant power when idle. The memory contents of DRAM are volatile, and thus require power even when idle to ensure data availability. Hence, it is not easy to reduce DRAM idle power. Flash memory is non-volatile, and does not require power when idle. However, flash memory is typically slow and has low data transfer speeds. Disks often continue to spin, even when idle, to provide quick access times for new data requests. Spinning down the disk can save power, but comes at the

cost of transition latency when spinning the disks back up for servicing new data requests.

- Is Race to Idle useful?
  The Race to Idle approach advocates working on customer requests continuously until all existing requests are served, and then transitioning to the idle state, until sufficient requests have accumulated. Race to Idle relies on having low idle power consumption, and having a workload that either has sufficiently long idle periods, or is delay insensitive. Web applications often have very short idle periods, and thus, are not well suited for Race to Idle. Batch workloads, on the other hand, are very well suited for Race to Idle because of their delay insensitivity. While Race to Idle is potentially beneficial for certain classes of workloads, it has been overshadowed by superior approaches such as DVFS [15].

- What are the right Voltage and Frequency settings for a given workload?
  DVFS is an approach that reduces server power consumption at the expense of reduced server speed by lowering the operating voltage and frequency. Thus far, DVFS has only been implemented for processors. However, there has been a lot of interest from the research community for DVFS in the memory subsystem [46, 50]. DVFS is well suited for applications that do not require the CPU or memory subsystem to be running at full speed. For example, an I/O-intensive application might only use the CPU for a very small fraction of time, and thus, would not be significantly impacted by lowering CPU voltage and frequency. Servers often have multiple voltage and frequency settings [69, 70]. The main challenge with DVFS is choosing the right voltage and frequency settings for a given application. Further, given the right settings, is DVFS superior to Race to Idle? These questions become even more challenging in a multi-server setting, where the number of choices for voltage and frequency settings (across all servers) grows exponentially [70].

1.3.2 Energy-efficient Server Design

Most servers are designed so as to provide great performance for a range of workloads. This flexibility often comes at the expense of not being energy-efficient. If an application has specific resource traits, such as being CPU-intensive or I/O-intensive, we can choose or build specific server designs that are a good fit for the application needs. Servers that are well suited to a given application can result in a system which has higher utilization, and consequently, is more energy-efficient. Further,

such servers can be architected to be power-aware, for example, by providing low-power inactive states, designing smarter processors, etc. We can realize the specific server design by either choosing from the range of available commodity machines (multi-core processor based servers to single-core mobile processor based devices) or designing a custom server (for example, a server with heterogeneous cores or a server with lots of Flash memory). The important challenges that come up when designing and deploying energy-efficient servers are as follows:

- Which is the best server architecture for a given workload?
  It is not always easy to determine the right server configuration for a given workload. Workloads often go through various execution phases where they exercise different components of the server. Further, the behavior of a workload can be dynamic and unpredictable, making it difficult to identify the workload's resource traits.

- Can we custom build power-aware servers?
  Modifying the server design, or building a new server design, involves a lot of challenges. When customizing the server design, we need to ensure that the functionality of the server is not compromised. For example, trying to add a new low-power inactive state ("sleep" state) might involve turning off the DRAM memory. This will lead to loss of DRAM contents. Is this acceptable to the application, or should the contents first be copied to a disk before putting the server to sleep? Alternatively, if the DRAM is to be kept turned on while the server sleeps, how can we provide access to the DRAM contents? These are difficult challenges, and often involve a tradeoff between lower power and reduced server functionality.

- Is the best server architecture cost-effective?
  Assuming that we identify the best server architecture for a given workload, it is not obvious that switching to this architecture will be cost-effective. For example, if the best architecture requires custom modifications to a commodity server, the cost overhead in making these modifications might render this switch infeasible.

1.3.3 Dynamic Server Provisioning

Dynamic server provisioning proposes to save power by only powering on the minimum amount of resources (servers) needed to satisfy the workload requirements.

The basic idea is to scale the number of servers in response to the workload demand. If the demand increases, additional servers are brought online. If demand decreases, unneeded servers are turned off, or put into low-power inactive "sleep" states, rather than being left idle. The advantage of sleep and off states is that a server consumes very little power in these states when compared to the idle state. However, a big disadvantage with the sleep and off states is that turning a server on from these states requires a significant amount of time (typically on the order of minutes [93, 129]). We refer to this transition time as the setup time:

Definition 1.3 (Setup Time)**.**
The setup time for a server is the time from when the server is turned on to the time when it is ready to execute customer requests.

Note that power (often peak power) is consumed during the entire setup time, even though no useful work is being done. Thus, there is also an energy penalty associated with setting up. Given these setup "costs," it is not obvious whether servers should ever be turned off.

In the context of data center power management, dynamic server provisioning advocates turning unneeded servers off to save power. However, in general, unneeded servers could also be repurposed for other applications (such as batch processing or data analytics), or loaned out to other customers in order to get more work done. Of course, if these servers were needed again in case workload demand increased, then they could be reclaimed after some setup time (to allow the server to free up). In the case of virtualized environments, the unneeded virtual machines (VMs) can simply be released back to the cloud to save on rental costs. Similar to server provisioning, there is a setup cost needed to create VMs or obtain them from the cloud. This setup cost can range anywhere from 30s – 1 minute if the VMs are locally created (based on our measurements using kvm [102]) or 2 – 10 minutes if the VMs are obtained from a cloud computing platform (see, for example, [13, 108]). Interestingly, the setup time for a VM can vary significantly depending on external factors such as time of day, location of host data center, etc [127].

In order to minimize the negative effects of setup time on performance, it is important to have a good estimate of workload demand. However, given the volatile nature of customer demands, workload prediction is often a difficult task.

We now describe the important challenges in dynamic server provisioning. We revisit these challenges in more detail in Chapter 3 and discuss the important prior work in dynamic server provisioning in Section 2.2.

- How to predict workload demand?
  Demand prediction is important for capacity planning. Most data center workload demands are very bursty in nature and often vary significantly during the course of a single day. This makes it challenging to predict future demand. Fortunately, most customer facing workloads do exhibit some (daily or weekly) periodic patterns [77, 20]. This allows us to estimate workload demand based on historic trends, provided that there is some correlation between historic data and current data. Unfortunately, sudden changes in demand are common in today's data centers. Important events, such as the September 11 attacks [113, 88], earthquakes or other natural disasters [186], slashdot effects [7], Black Friday shopping [44], or sporting events, such as the Super Bowl [145] or the Soccer World Cup [20], are common causes of load spikes. While some of the above events are predictable, most of them cannot be predicted in advance. This makes it even more challenging to predict workload demand.

- How many servers?
  Assuming we have some prediction for the future workload demand, how many servers do we need to handle this demand? Converting workload demand to capacity requires information on the performance SLA (see Definition 1.2). Given the SLA and predicted workload demand, we can use either performance modeling or experimental studies to determine the required number of servers to successfully handle the estimated demand. However, since actual demand can vary from the predicted demand, we need to allocate some spare capacity in the system. The spare capacity can either be in the form of additional servers, or servers that are in sleep modes. The amount of spare capacity needed will depend on the expected deviation between actual demand and predicted demand, and the SLA. In general, determining the right amount of spare capacity is a very difficult question. This is further complicated by setup times, which can severely impact response times. The problem is further exacerbated in the presence of stateful servers, due to the temporary unavailability of data following the addition or removal of a stateful server.

- How useful are sleep states?
  Since the idle power consumption of DRAM and disk cannot be easily reduced, an orthogonal approach is to transition the entire server to a low-power inactive "sleep" state when there is no work to do. Desktops, laptops, and mobile devices are all equipped with sleep states that allow the power consumption to be near-zero when there is no work to do. While sleep states result in a much lower power consumption than idle power, they require a significant transition

latency to go back to the active state [131, 68]. Thus, it is not obvious whether sleep states are useful or not. Further, there are often multiple sleep states that one can transition to when idle. For example, in the S3 state (Suspend to RAM), the CPU is turned off but the memory and disk subsystems remain on, whereas in the S4 state (Suspend to Disk), the CPU and the memory subsystem are turned off, but the disk remains on [135]. The "off" state can also be considered as a sleep state with zero power. The different sleep states differ in their sleep power consumption and their setup time (see Definition 1.3). Given their different characteristics, it is not obvious which (if any) sleep states are useful for a given application. Also, when should the server transition into and out of the sleep states?

1.3.4 Consolidation and Virtualization

Consolidation takes dynamic server provisioning one step further and allows different application instances to be colocated on the same physical server. The basic idea is to consolidate the workload from several, possibly under-utilized, servers onto fewer servers. This allows the unneeded servers to be turned off, resulting in lower power consumption, and higher system utilization. We now describe the important challenges encountered when consolidating workloads:

- How should virtual machines be consolidated on physical machines?
  Different applications have different resource usage traits. This makes it challenging to colocate different application instances together on a physical server. For example, it might be more beneficial to colocate a CPU-intensive application with a memory-intensive application rather than colocating two CPU-intensive applications. Identifying the resource usage patterns of applications is often very difficult because of the dynamic nature of data center workloads. There is also the concern that colocation might expose critical applications to security risks because of other malicious applications.

- How to allocate resources fairly in a cloud environment?
  When consolidating multiple applications with different resource needs, fairness becomes an important metric. The data center operator has to ensure that applications do not starve because of other, more demanding, applications.

Chapter 2

Thesis Overview

Power management in data centers is a very broad topic, as we have seen in Section 1.3. In this thesis, we focus on *dynamic server provisioning and the challenges associated with it*. This chapter provides an overview of the contents of this thesis.

We start in Section 2.1 with a brief discussion of dynamic server provisioning. In particular, we provide the motivation for choosing dynamic server provisioning as our approach for data center power management in Section 2.1.1. We then discuss the challenges involved in dynamic server provisioning in Section 2.1.2, and briefly discuss the new research needed for addressing these challenges in Section 2.1.3. We discuss prior work related to dynamic server provisioning in Section 2.2 and highlight the challenges that have been overlooked by these works. We then talk about the scope of this thesis in Section 2.3, specifically, the metrics that we consider in this thesis in Section 2.3.1, and our implementation testbed in Section 2.3.2. Given the metrics and our testbed, we then list the research questions that this thesis will address in Section 2.3.3. Finally, in Section 2.4, we differentiate our work from prior work and list the contributions of this thesis.

2.1 Dynamic Server Provisioning

The approaches used for dynamic power management in this thesis can best be classified as dynamic server provisioning approaches. In general, we are interested in minimizing the total number of servers needed at any point of time for meeting the response time SLAs (see Definitions 1.1 and 1.2) of an application. In this thesis, we use the term "servers" to refer not only to physical servers, but also to other resources

in general such as virtual machines, or even processor cores on a given physical server. Since dynamic server provisioning involves adding and removing servers, we deal with the associated challenges such as setup times (see Definition 1.3), unpredictability in workload demand, and availability of data. We discuss these challenges in Chapter 3.

In this section we briefly provide the motivation for choosing dynamic server provisioning as our approach for data center power management. We then discuss the factors that make dynamic server provisioning difficult, and how prior work has handled (or sidestepped) these difficulties. We end this section by outlining the new research required to fully realize the potential of dynamic server provisioning, thus making a case for this thesis.

Note that while this thesis focuses on dynamic server provisioning in data centers, the techniques discussed herein can easily be extended to dynamic capacity provisioning problems in the cloud. For example, consider private clouds, where unneeded servers can be repurposed for *valley-filling*, that is, doing other useful work instead of being turned off or put to sleep. This "other" useful work can be either maintenance or background work, or batch workload such as data analytics. Another extension of dynamic provisioning is capacity management in community clouds, where unneeded servers can be given away to other trusted groups to increase the total throughput. In both of the above examples, if these "donated" servers were needed again in case workload demand increased, then they could be reclaimed after some setup time. This setup time allows the other applications running on these servers to be migrated or terminated in a graceful manner. Yet another extension of dynamic provisioning is for cost management in public clouds, where virtual machines (VMs) are leased to users who typically pay an hourly rent for using the VMs. In such settings, users are interested in meeting their performance needs while minimizing their rental expense. Dynamic server provisioning helps the users minimize their required VMs, and the unneeded VMs can be released back to the cloud provider to reduce rental costs. If additional VMs are required at a later time, they can be obtained from the cloud by incurring a setup cost, which is typically on the order of minutes [13, 108].

2.1.1 Motivation behind dynamic server provisioning

As we stated in Section 1.2, power is an expensive resource in data centers. Unfortunately, *a lot of power in data centers is actually wasted*. One of the big reasons for this waste is low server utilization. Servers in a data centers are often left "always on," leading to only 10–30% server utilization [21, 26]. In fact, [167] reports that the average data center server utilization is only 18% despite years of deploying

virtualization aimed at improving server utilization. From a power perspective, low utilization is problematic because servers that are on, while idle, still utilize 60% or more of peak power. Further, servers that are on necessitate support from other IT infrastructure including the cooling, networking, and storage systems. Thus, idle servers also lead to indirect power consumption. Given the importance of reducing power consumption in data centers and the fact that data centers have low server utilization, we conclude that dynamic server provisioning is an attractive solution for data center power management.

2.1.2 Why is dynamic server provisioning difficult?

Dynamic server provisioning has a lot of potential for reducing power consumption in data centers. However, *there are many challenges that hinder the successful deployment of dynamic server provisioning*. While dynamic server provisioning has received a lot of attention recently (see Section 2.2 below), most of the existing research has ignored, or carefully sidestepped, these important challenges.

For example, *prior work often assumes that dynamic server provisioning decisions only need to be taken once every epoch* [41, 191, 40] (on the order of tens of minutes). At this coarse granularity, authors ignore fine-grained variations in load, which are all too common in data center workloads [73, 60, 61], and which have a significant negative impact on the potential for power savings [188].

Another *important problem that is very often ignored is the setup time required to provision new servers*. The setup time for servers is typically on the order of minutes [93, 129]. As we show in Chapter 3, setup times significantly impact response times, and thus cannot be ignored when dynamically provisioning servers. We are only aware of a handful of recent publications ([188, 107]), apart from our own, that take setup times into account. There are some publications ([121, 60, 120, 41, 129]) that partially take setup time into account by assigning a penalty (usually to the operating cost) every time a server is switched on. Such penalties ignore the adverse effects of setup time on performance.

A common assumption in server provisioning is that workload demand can be predicted [189, 86, 114, 141, 57, 3, 190, 196, 53]. Unfortunately, this is not always the case. As we show in Chapter 3, *even for workloads with periodic trends, there is often a portion of the demand that is unpredictable. Ignoring this unpredictable portion can lead to a steep increase in response times.* Further, demand can sometimes spike unpredictably, causing response times to shoot up. *Most prior work in dynamic server provisioning ignores load spikes by designating them as rare events.*

Finally, *dynamic server provisioning in the presence of stateful servers is another problem that has often been overlooked*. This is a difficult problem because of the temporary data unavailability that follows the addition or removal of a stateful server, such as a caching tier server. The only prior work in this area that we are aware of focuses on scaling the number of replicas in distributed storage systems [14, 170].

2.1.3 The need for this thesis

The above mentioned challenges limit the applicability and benefits of dynamic server provisioning in data centers. As we show in Section 2.2 below, prior work has often carefully evaded these challenges at the cost of reduced benefits. For example, by only making provisioning decisions at very coarse time scales, prior work avoids setup times. However, this coarse-grained provisioning makes the system vulnerable to load spikes and short-term fluctuations. To deal with these issues, prior work often resorts to defensive tactics, such as over-provisioning, load-shifting, or admission control. Such corrective measures, though useful, limit the potential for power savings, and can hurt response time. *In order to realize the full potential of dynamic server provisioning, we must overcome the associated challenges.*

Addressing the challenges associated with dynamic server provisioning requires new research, both in theory and systems. For example, the fact that setup costs (a.k.a. switching costs or wake-up penalties) negatively impact response times is common knowledge [26, 173, 86] among researchers in dynamic provisioning. However, no closed-form analysis exists for multi-server systems with setup times, even though a single-server with setup times was completely analyzed back in 1964 [194]. Thus, there is a need for new theoretical research to understand the effects of setup times. Another example is dynamic provisioning of the caching tier. Scaling the caching tier can provide significant savings in power and cost, since caching tier server are typically [128] equipped with massive amounts of expensive and power-hungry DRAM. However, to the best of our knowledge, there has been no prior work on dynamic scaling of the caching tier. Thus, there is a need for new systems research to implement and evaluate novel dynamic provisioning approaches for the caching tier. *This thesis provides new theoretical and systems research contributions that address the open challenges in dynamic server provisioning, such as those discussed above.*

2.2 Prior Work in Dynamic Server Provisioning

In this section we discuss prior work in data center power management that specifically employs dynamic server provisioning. When applicable, we highlight the ideas from prior work that this thesis builds upon. We also highlight the challenges addressed, or overlooked, by the prior work. We revisit some of this prior work in later chapters when we focus on specific research problems. For a discussion of related work employing other techniques for data center power management, see Chapter 8.

Most dynamic server provisioning approaches can be categorized into two types: predictive and reactive (or control-theoretic). Predictive approaches, e.g., [107, 40, 152, 36, 120, 121], aim to predict what the request rate will be in the future, so that they can start turning on servers now if needed. Reactive approaches, e.g., [189, 86, 114, 141, 57, 3, 190, 196, 53], all involve reacting immediately to the current request rate (or the current response time, or current CPU utilization, or current power, etc) by turning servers on or off. There are also hybrid approaches, e.g., [41, 28, 191, 174, 76, 173, 60, 61], that combine ideas from predictive and reactive approaches. There are also approaches that are neither predictive nor reactive, and instead, rely on the application to express its resource requirements [58, 39, 187, 91]. We categorize these approaches as *decentralized* approaches. Finally, there are also approaches for dynamic provisioning in the operations research literature, e.g., [120, 121, 90]. While the system model in these operations research works is different from our data center setting, there are certain conceptual similarities. We now discuss in detail the relevant prior work in dynamic server provisioning. We omit discussion of some of the prior work (such as [119, 196, 191]) where setup times are negligible, since one of the challenges we wish to address explicitly in this thesis is dealing with setup times.

2.2.1 Predictive approaches

Krioukov et al. [107] use various predictive policies, such as Last Arrival, Moving Window Average, Exponentially Weighted Average, and Linear Regression, to predict the future request rate (to account for setup time), and then accordingly add or remove servers from a heterogenous pool. The authors evaluate their dynamic server provisioning policies by simulating a multi-tier web application. The authors find that Moving Window Average and Linear Regression work best for the traces they consider (Wikipedia.org traffic), providing significant power savings over conservative, static approaches. We make use of this result in Chapter 4 to narrow the set of predictive policies that we implement for comparison against our proposed policies.

Chen et al. [40] use auto-regression techniques to predict the request rate for a seasonal arrival pattern, and then accordingly turn servers on and off using a simple threshold policy. The authors evaluate their dynamic server provisioning policies via simulation for a single-tier application. They find that their dynamic server provisioning policy performs well for periodic request rate patterns that repeat, say, on a daily basis. Some of the ideas behind our dynamic server provisioning policy presented in Chapter 4 are similar to those used by Chen et al. Specifically, the idea of index-based routing, which helps to concentrate load on fewer servers, is common to both works.

In general, predictive approaches are very successful when dealing with periodic or seasonal workloads. However, as we show in Chapter 4, these approaches fail when the workload is bursty and unpredictable, or when the workload demand suddenly increases: It is clearly hard to predict what will happen in the future when demand is bursty and future arrivals are unknown.

2.2.2 Reactive approaches

Horvath et al. [86] employ a reactive feedback mechanism to provision capacity for a multi-tier web application. In particular, the authors monitor server CPU utilization and response times, and react by adding or removing servers based on the difference between observed response time and target response time. The authors evaluate their reactive approach via implementation in a multi-tier setting. The authors also study the effect of using multiple sleep states in servers, and conclude that using sleep states in conjunction with the traditional off state can significantly improve energy efficiency. In our work in Chapter 4, we also consider monitoring CPU utilization and/or response times to provision servers. However, we discard these metrics in favor of a more robust metric – number of active requests in the system. Also, in Chapter 4, we evaluate the use of sleep states for dynamic server provisioning, and come to the same positive conclusion as Horvath et al.

Wang et al. [189] also employ a reactive feedback mechanism to manage the power-performance tradeoff in multi-tier systems. The authors use DVFS along with capacity provisioning to react to degradation in observed response times. Similar to our work in Chapters 3 and 4, Wang et al. leverage queueing theoretic results to help guide their system. However, the models used in Wang et al. are much simpler than ours, and do not take setup time into account.

Abdelzaher et al. [3] use a control-theoretic approach to provision resources to applications in a multi-tier architecture. The authors use a queueing-theoretic model

to predict response times. Their closed-loop approach then manages the tradeoff between the number of applications hosted on a machine and the amount of resources (CPU or memory) allocated to each application based on the estimates of response time and power consumption. The authors' use of queueing theory to guide dynamic provisioning is similar to our efforts in Chapters 3 and 4. However, just as in Wang et al. [189] above, the queueing models employed by Abdelzaher et al. do not take setup time into account.

While reactive approaches often lead to robust dynamic server provisioning solutions, they can be inadequate for meeting response SLAs when the setup time is high. As we show in Chapter 4, this is because the benefits of the increased capacity only take effect after the setup time.

2.2.3 Hybrid approaches

Hybrid approaches typically work by relying on predictions to capture the long-term trends in workload demand, and then employing reactive techniques to handle the unpredictable short-term fluctuations in demand.

In Urgaonkar et al. [173] and Gandhi et al. [60], the authors consider workloads where the demand is divided into two components, a long-term trend which is predictable, and short-term variations which are unpredictable. The authors use predictive approaches to provision servers for long-term trends (over a few hours) in request rates, and then use a reactive controller to react to short-term variations in request rate. In our work, we assume unpredictable demand. However, as in the hybrid approaches above, we start with a reactive controller to handle the unpredictable short-term variations in load.

Hybrid approaches combine the strengths of predictive and reactive approaches, and are superior to purely reactive or purely predictive approaches [60, 61]. However, hybrid approaches continue to suffer from the shortcomings of predictive and reactive approaches, albeit to a lesser extent. Specifically, the reactive controller is still vulnerable to setup times. However, because of the predictive component, the reactive controller now only needs to handle the short-term fluctuations in workload demand, which is often only a small fraction of the total demand [60, 61].

2.2.4 Dynamic provisioning for stateful servers

All of the above prior work assumes stateless servers. We now discuss important prior work in dynamic provisioning of stateful servers.

Amur et al. [14] and Thereska et al. [170] consider the problem of organizing and scaling data replicas such that at least one copy of the data is always on. This allows the remaining data nodes to be scaled down without affecting availability. Writes directed at off servers are instead written to other servers (write offloading), and later reorganized to conform to the desired data layout. In general, write offloading [139] has been used to redirect requests to otherwise idle disks to increase periods of idleness, allowing these disks to be spun down to save power. In our work in Chapter 6, we also use the concept of offloading requests to allow otherwise idle servers to be turned off. However, we use this idea for read requests in a novel way that allows hot data items to be moved from otherwise idle servers to other servers, before the idle servers are turned off.

Prior work in dynamic provisioning for stateful servers has only focussed on distributed storage systems [14, 115, 170, 179, 161]. The main contribution in this area has been the development of different mechanisms and data layouts for replicated storage systems that allow storage servers to be turned off while maintaining availability. To the best of our knowledge, there has been no prior work on dynamic provisioning of caching tier servers.

2.2.5 Decentralized dynamic server provisioning approaches

One of the major problems with centralized approaches, such as the ones discussed above, is that they do not scale well. Thus, recent literature has explored fully or partially decentralized approaches based on applying concepts from economic theory (see [58] for a broad discussion of such approaches). Chase et al. [39] develop a bidding approach whereby resource units can be bought with virtual currency, and customers bid for these resource units. The objective for the central agent is to maximize its profit at each time interval by "selling" the available resource units to the highest bidders. A similar bidding framework was also used in [187] to "sell" idle CPU cycles to customers in a distributed computing environment and in [91] to lease network resources to customers.

In general, decentralized approaches suffer from certain shortcomings. Decentralized controllers typically have less information than centralized controllers, and are thus limited in their potential for power savings. Further, they require the application to express its resource requirements, which is often non-trivial to compute.

2.2.6 Dynamic provisioning approaches in operations research

The problem of dynamic server provisioning also comes up in operations research in the context of staffing employees in call centers and also in inventory management. In call center staffing, the servers are employees, who require a salary (power) when they are working. The call center manager can bring in additional employees when the demand increases, at the expense of a setup cost (time to reach work and extra pay). Unfortunately, most of the analytical work in call center staffing has either ignored setup costs or assumed stationary demand. One exception to this is [97]. In [97], the authors consider the problem of dynamic staffing based on knowing the distribution of demand so as to upper bound the probability of a customer finding all servers busy on arrival. The authors show that the optimal policy suggests provisioning for the sum of the mean load and the square-root of the mean load; that is, provision spare servers in proportion to the square-root of the mean load. This is known as the "square-root staffing" rule. Unfortunately, this rule requires knowledge of the future load. Our work in Chapter 4 presents a new dynamic server provisioning policy, *AutoScale*, that converges to "square-root staffing" without knowing the future load.

Within inventory management, the problem of dynamic provisioning takes the form: how much inventory should one maintain so as to minimize the total cost of unused inventory (holding cost, in our case, idle power) and queueing time. A popular dynamic policy that is employed in such cases is known as Make to Order, which is similar in nature to reactive approaches. In inventory management, a common assumption is that inventory is produced sequentially. In the context of dynamic server provisioning, this assumption translates to allowing at most one server to be in setup at any time. We refer to this model as *staggered setup*. The inventory management analysis [6, 22] on staggered setup has not resulted in any closed-form solutions for the response time or power. We provided the first closed-form analysis of staggered setup in [66, 67, 68], and showed that the distribution of response time for multi-server systems with staggered setup has a nice decomposition property. Later, in [62], we relaxed the staggered setup requirement, and provided the first closed-form analysis of multi-server systems with setup costs. We discuss this analysis in Chapters 3 and 7.

All of the prior work in operations research discussed above looks at the average behavior of the system. There is also a lot of prior work in power management under operations research that looks at the worst-case behavior of the system [121, 120, 90]. Since our focus in this thesis is not on worst-case analysis, we omit the discussion of these works.

2.3 Scope of this Thesis

The scope of this thesis is limited to dynamic server provisioning approaches for data center power management. In particular, this thesis aims to address the important challenges discussed in Section 2.1.2. As we have seen in Section 2.2, prior work in dynamic server provisioning has often dodged these challenges. The specific research questions that we address in this thesis are listed in Section 2.3.3, below. The dynamic server provisioning solutions presented in this thesis combine theoretical research and systems research. Our theoretical research uses performance modeling and novel queueing-theoretic analysis to understand the factors affecting dynamic server provisioning, such as setup times and unpredictable workload demand. Our analysis allows us to estimate the effect of these factors on the specific response time and power metrics that we care about. These metrics are discussed below in Section 2.3.1. Based on our theoretical insights, we design our dynamic server provisioning solutions. All of our proposed solutions in this thesis are experimentally evaluated using realistic workloads and application traces. The details of our implementation testbed are discussed in Section 2.3.2 below.

2.3.1 Metrics

We consider two different sets of metrics in this thesis. The first set of metrics we are interested in are those that quantify performance. As mentioned in Chapter 1, we use response time (see Definition 1.1) as our performance measure. Given a trace of customer requests, we are often concerned about the mean response time, T_{avg}, of the requests over the duration of that trace. Formally, T_{avg} is defined as:

Definition 2.1 (T_{avg}).
For a give trace, we define T_{avg} as the mean response time (in seconds or milliseconds) for requests that complete during the course of the trace.

Nowadays, it is more common to look at higher percentiles of response time rather than just the mean. The motivation behind looking at higher percentiles is to ensure that most of the customers experience low response times and only a small fraction, if any, of the customers experience high response times. Thus, in addition to T_{avg}, we are also concerned about the 95th percentile of response times, T_{95}. It would be equally easy to look at the 90th or the 99th percentile of response times. Our choice of 95 is motivated by recent studies [173, 107, 132, 48] which indicate that

95th percentile response times are typical for quantifying performance. Formally, T_{95} is defined as:

Definition 2.2 (T_{95})**.**
For a give trace, we define T_{95} as the 95th percentile of response times (in seconds or milliseconds) for requests that complete during the course of the trace.

We sometimes also care about the instantaneous T_{95} values, which we define as:

Definition 2.3 (Instantaneous T_{95})**.**
Instantaneous T_{95} is defined as the T_{95} for requests that completed during the past one second.

The second set of metrics we are interested in are those that quantify power consumption, or more generally, resource consumption:

Definition 2.4 (P_{avg})**.**
For a give trace, we define P_{avg} as the mean power consumption (in watts) of the servers during the course of the trace.

Definition 2.5 (N_{avg})**.**
For a give trace, we define N_{avg} as the mean number of active servers employed during the course of the trace. This includes servers that are busy, idle, or in setup, but does not include servers that are off.

The goal in this thesis is to optimize some function of the response time metric and the resource consumption metric. Unless stated otherwise, the goal will be to minimize P_{avg} while ensuring that T_{95} remains below a certain threshold.

Sometimes, for optimization, we are interested in a joint function of response time and power consumption. We use the popular Performance-per-Watt (PPW) metric [72, 64] for these purposes. We formally define PPW as:

Definition 2.6 (PPW)**.**
For a give trace, PPW is defined as the inverse of the product of T_{95} and P_{avg}, and is given mathematically by:

$$PPW = \frac{1}{T_{95} \cdot P_{avg}} \tag{2.1}$$

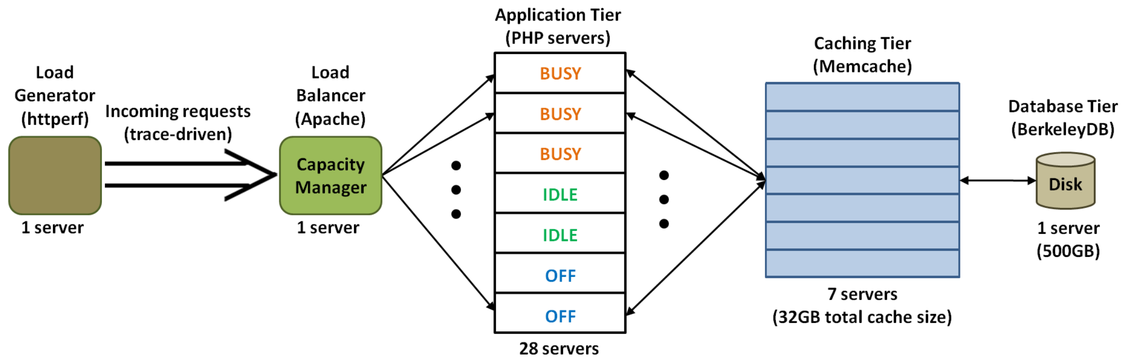


Figure 2.1: Our implementation testbed.

2.3.2 Implementation testbed

Figure 2.1 illustrates our multi-tier testbed, consisting of 38 Intel Xeon processor-based servers, each equipped with two quad-core processors. Our testbed consists of a load generator, a load balancer (and capacity manager), a stateless application tier, and stateful caching and database tiers. Recall from Section 1.1 that stateless servers do not contain any data, and stateful servers contain data required to serve customer requests.

We employ one of the servers as the front-end load generator running httperf [138]. Httperf allows us to generate trace-driven arrivals for our experiments. Another server is employed as the front-end load balancer running Apache, which distributes requests from the load generator to the application tier servers. We modify Apache on the load balancer to also act as the capacity manager, which is responsible for adding and removing servers (by turning servers on and off). Another server is used to store the entire data set, a billion key-value pairs, on a 500GB BerkeleyDB [146] database. Seven servers are used as memcached servers, each with 4GB of memory for caching, for a total cache size of 28GB. The remaining 28 servers are employed as PHP servers that parse the incoming requests and collect the required data from the back-end memcached and database servers. Our ratio of application servers to memcached servers is close to the typically reported ratio of 5:1 [55].

We employ server provisioning for the application tier and caching tier servers. In our testbed, the load balancer acts as the capacity manager, and scales capacity by turning servers on and off. We can easily extend the functionality of the capacity manager to add and remove virtual machines in a cloud environment. We remotely turn servers on and off by communicating with the power distribution unit (PDU) via

the SNMP [130] protocol. We monitor the power consumption of individual servers by reading the power values off of the PDU. The idle power consumption for our servers is about 140W (with C-states enabled) and the average power consumption for our servers when they are busy or in setup is about 200W.

In our experiments, we observe the setup time for the servers to be about 260 seconds. Most of this time is spent in copying the OS image over the network and booting the OS, initializing the RAID controller, and in establishing network connectivity. We also examine the effects of lower setup times that could either be a result of using sleep states (which are prevalent in laptops and desktop computers, but are not well supported for server architectures yet), or using virtualization to quickly bring up virtual machines. We replicate this effect by not routing requests to a server if it is marked for sleep, and by replacing its power consumption values with the sleep state power values. When the server is marked for setup, we wait for the setup time before sending requests to the server, and replace its power consumption values during the setup time with 200W.

We design a key-value workload to model realistic multi-tier applications such as the social networking service, Facebook, or e-commerce services like Amazon [48]. Each generated request is a PHP script that runs on the application server. A request begins when the application server requests a value for a key from the memcached servers. The memcached servers provide the value, which itself is a collection of new keys. The application server then again requests values for these new keys from the memcached servers. This process can continue iteratively. We set the number of iterations to correspond to an average of roughly 3,000 key-value pairs per request, which translates to a mean request size of approximately 120 ms, assuming no resource contention. Each key-value request in our workload is a read operation (or GET operation). The request size (or service time) distribution is highly variable: the service time of the largest request is roughly 20 times the service time of the smallest request. We can also vary the distribution of key-value requests by the application server. In this chapter we use the Zipf [143] distribution, whereby the probability of requesting a particular key-value pair varies inversely as a power of that key.

We use the above specified configuration for our implementation testbed in most of our experiments. However, we sometimes modify the testbed to explore specific settings. In such cases, we will describe the modifications made to the above testbed.

2.3.3 Research questions addressed by this thesis

The research in this thesis can be categorized into two parts. The first part, presented in Chapter 3, aims to understand the challenges involved in dynamic server provisioning, and the effects they have on response time and power. In particular, we address important research questions pertaining to setup times, unpredictability in workload demand, and data availability:

1. What is the effect of setup time on response time and power?
2. How to analyze multi-server systems with setup times?
3. Can we completely predict workload demand?
4. How do load spikes affect response time?
5. How much can we save by scaling the caching tier?
6. What happens to response time when we add or remove a caching server?

The above questions are fundamental in nature and are not specific to a given application. We thus use theoretical analysis to answer the above questions.

The second part of our research, presented in Chapters 4, 5, and 6, provides practical dynamic server provisioning solutions that overcome the above problems. The important questions concerning the implementation of dynamic server provisioning that this thesis addresses are:

1. How many servers are needed to handle the incoming workload?
2. When should servers be turned on/turned off/left idle/put to sleep?
3. Which sleep states are "useful" for a given application?
4. What policy should be used to route incoming requests to servers?
5. How do we handle dynamic changes not just in request rate but also dynamic changes in request size and server speed?
6. How do we handle load spikes?
7. How can we dynamically provision the caching tier?

We address the above questions in the context of our implementation testbed described in Section 2.3.2.

2.4 Novelty and Contributions

In order to answer the questions listed above in Section 2.3.3, we first analyze the problems that hinder dynamic server provisioning. We use theoretical performance modeling and queueing theory to estimate the effect of these problems on response time and power. Armed with an understanding of their effects, we then present practical dynamic server provisioning solutions that overcome these problems and lower power consumption in data centers. We now give a brief summary of the contributions of this thesis.

- We start in Chapter 3 with a detailed study of the problems that hinder dynamic server provisioning in data centers. Specifically, we examine setup costs (Section 3.1), uncertainty in workload demand (Section 3.2), and provisioning in the presence of stateful servers (Section 3.3). In order to analyze the above problems, we use a mix of theoretical analysis and experimentation. Our analysis reveals several interesting results. For example, we find that *the adverse effects of setup time on response time decrease in severity with an increase in the size of the system*. Based on our findings, we provide practical solutions for dynamic server provisioning in data centers.

- In Chapter 4 we consider the problem of *dynamic provisioning for stateless servers in a multi-tier data center with setup costs and completely unpredictable demand*. We first show that existing dynamic server provisioning policies fail in the presence of setup times and unpredictable demand. We then present *AutoScale*, our robust dynamic server provisioning policy that *provably* handles unpredictable changes in request rate, request size, and server speeds. *AutoScale* is very different from existing dynamic server provisioning approaches, in that it does not try to predict the future request rate. Instead, *AutoScale* makes the case that it often suffices to simply be conservative when scaling down capacity. We demonstrate, via implementation on our 38-server testbed (see Section 2.3.2), that *AutoScale* successfully meets response time SLAs while significantly reducing power consumption even for bursty, unpredictable workloads. *AutoScale* also helps to reduce expenses when renting VMs from the cloud by minimizing the number of VMs needed to meet response time SLAs.

- In Chapter 4 we also consider the problem of *which sleep states are useful for a given application*. A sleep state can be defined by its setup time and the power consumed when in sleep. We tune our testbed so as to replicate the effects of a sleep state, and evaluate the behavior of different dynamic server

provisioning policies, including *AutoScale*, using more than a hundred different sleep states. Our results show that sleep states with lower sleep power are preferable for workloads whose demand varies slowly over time, whereas sleep states with lower setup time are preferable for bursty workloads. However, for particularly bursty traces, such as traces with load spikes, we find that even a 20s setup time does not suffice. We also present an orthogonal evaluation of server provisioning policies under lower idle power consumption.

- In Chapter 5 we consider the problem of *provisioning multi-tier data centers in the presence of load spikes*. We first show, via implementation, that even a small setup time (5 seconds) can lead to a steep increase in response time when faced with load spikes. We then present *SoftScale*, an approach to handling load spikes in multi-tier data centers. *SoftScale* works by opportunistically stealing resources from other tiers to alleviate the bottleneck tier, even when the tiers are carefully provisioned. We demonstrate, using a range of workload traces, that *SoftScale* allows the system to meet stringent response time SLAs even if workload demand doubles instantaneously. *SoftScale* is especially useful during the transient overload periods when additional capacity is being brought online. Importantly, *SoftScale* can be used in conjunction with existing dynamic server provisioning policies.

- In Chapter 6 we consider the problem of *provisioning for the stateful caching tier*. Caching tier servers are often provisioned with massive amounts of expensive and power-hungry DRAM. Thus, dynamic provisioning of the caching tier can lead to huge savings in cost and power. However, dynamic provisioning of the stateful caching tier is complicated because, apart from response time and power, we also have to worry about data availability. We present *CacheScale*, a novel dynamic provisioning approach for the caching tier that meets response time SLAs while scaling caching tier capacity based on changes in load. *CacheScale* ensures that scaling the caching tier does not significantly impact hit rate by proactively redistributing "hot" data items. *CacheScale* does this without requiring access to the elusive least-recently-used list of cached items. This makes it very easy to deploy *CacheScale* on any caching tier.

- In Chapter 7 we present our novel analytical technique, Recursive Renewal Reward (RRR). The RRR technique allows us to predict the effects of setup time on response time and power in a multi-server system. In doing so, we provide *the first analysis of multi-server systems with setup times*. The RRR technique is highly intuitive and very easy to apply. Mathematically, RRR

allows us to analyze Markov chains with a repeating structure. This class of Markov chains is widely used to model a range of multi-server computer systems. We thus anticipate that RRR will prove useful to other researchers in analyzing many new problems.

Chapter 3

Challenges in Dynamic Server Provisioning

In this chapter we discuss the challenges in dynamic server provisioning, and analyze their effects on response time and power consumption. The three most important challenges that, we believe, hinder the deployment of dynamic server provisioning policies in data centers are:

1. Setup costs;
2. Uncertainty in workload demand; and
3. Presence of stateful servers.

We analyze each of these challenges in detail. In particular, we start in Section 3.1 by addressing setup costs. The analysis of multi-server systems with setup costs has been a long-standing open problem in queueing theory. In Section 3.1.1, we provide the first closed-form analysis of setup costs. We use our analysis to study the effect of scale (data center size) on setup times in Section 3.1.2. Next, in Section 3.2, we address uncertainty in workload demand. We analyze workload demand traces obtained from a few data center workloads and from publicly available sources in Section 3.2.1. We then consider load spikes (abrupt changes in load) in workload demand in Section 3.2.2. We use our implementation testbed to investigate the detrimental effects of load spikes. Finally, in Section 3.3, we consider dynamic provisioning of stateful (caching tier) servers. Using our multi-tier testbed, we experimentally analyze the effects of adding and removing a caching server in Section 3.3.1. We summarize the findings of this chapter in Section 3.4.

3.1 Setup Costs

Servers in a data center consume peak power when they are servicing customer requests, but still consume about 60% [26] of that peak power when they are idle. To save power, idle servers can be turned off. However, there is a setup cost involved in turning a server back on. This setup cost is in the form of a time delay (typically on the order of minutes [93, 129]), *and* a power penalty, since the server typically consumes peak power during the entire setup time.

Surprisingly, for all the importance of setup times, very little is known about their analysis. A single-server system with setup times was analyzed in 1964 by Welch [194]. The analysis of a multi-server system with setup times, however, has remained elusive. Using queueing theory, we now provide the first analysis of multi-server systems with setup costs.

3.1.1 Analysis of setup costs

In order to model a multi-server system, we use concepts from queueing theory [160, Chapter 8]. Specifically, we assume that the incoming requests arrive to the system according to a Poisson process with mean request rate, λ. Although we assume a constant mean request rate, the Poisson process allows us to study the effect of short-term variations in request rate. We consider a k server homogeneous system with each server running at a speed of μ requests/sec, where request sizes are assumed to be exponentially distributed with mean $\frac{1}{\mu}$. The servers serve requests on a first-

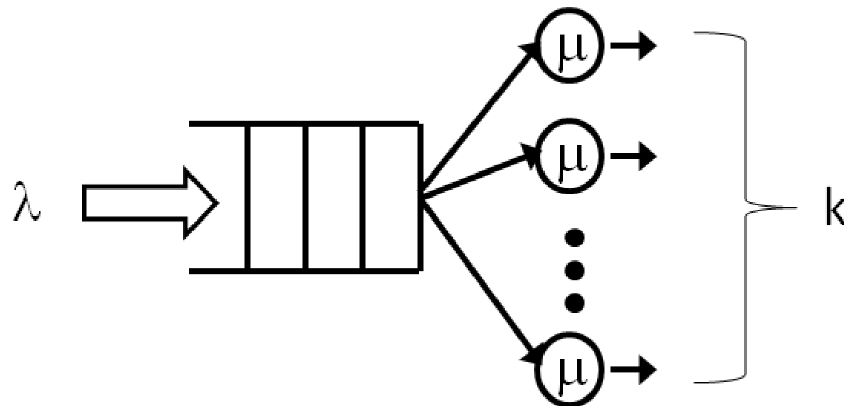


Figure 3.1: Our M/M/k queueing model for a multi-server system.

come-first-serve basis. This k server system is called an M/M/k queueing model, and is shown in Figure 3.1. Given the mean request rate, λ, and total mean service rate, $k\mu$, we define load, or utilization, to be $\frac{\lambda}{k\mu}$. Load gives us an indication of the utilization of the system, and is thus a helpful quantity to consider.

Definition 3.1 (Load)**.**
For an M/M/k system with mean request rate λ and mean service rate (for each server) μ, the load of the system is defined as the mean utilization of the system, and is given mathematically by:

$$Load = \frac{\lambda}{k\mu} \tag{3.1}$$

For stability, we assume that $\frac{\lambda}{k\mu} < 1$, that is, the system is not overloaded.

Model for setup times

In order to analyze setup times, we develop a new multi-server queueing model – the M/M/k/setup. In this model, each of the k servers is in one of three states: off, on (being used to serve a request), or setup. When a server is on or in setup, it consumes peak power of P_{peak} watts. When a server is off, it consumes zero power. We assume that when servers are not in use, they are immediately turned off to save power. However, turning on an off servers requires some setup time. We assume that setup times are exponentially distributed with mean rate α, which is the inverse of the mean setup time, t_{setup}. The full M/M/k/setup model is described in detail in Chapter 7.

Our goal is to derive the mean response time, T_{avg}, and the mean power consumption, P_{avg}, for the M/M/k/setup model. In order to do this, we track the number of active requests in the system and the number of on servers. Given our M/M/k/setup model, we can easily track these quantities using a Markov chain [160, Chapter 6]. The Markov chain for our M/M/k/setup system is shown in Figure 3.2. Each state is denoted by the pair (i, j), where i is the number of on servers, and j is the number of requests in the system. Thus, the number of servers in setup is $\min\{j - i, k - i\}$. Note that the Markov chain is infinite in one dimension. The complexity of this 2-dimensional, infinite Markov chain makes it very difficult to analyze.

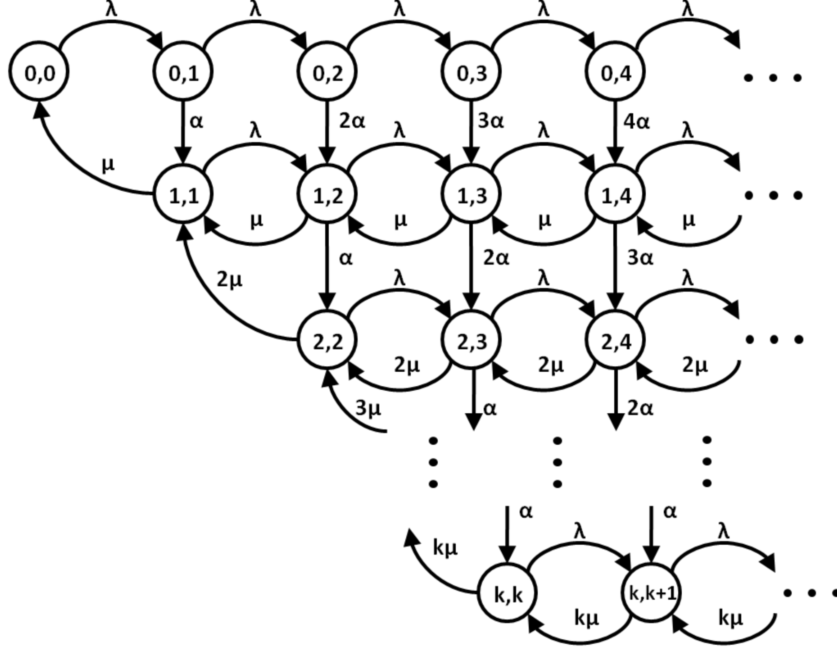


Figure 3.2: M/M/k/setup Markov chain. Each state is denoted by the pair (i, j), where i is the number of on servers, and j is the number of requests in the system. The number of servers in setup is $\min\{j - i, k - i\}$.

Our new analysis technique

Our analysis of the M/M/k/setup is made possible by our development of a new technique, Recursive Renewal Reward (RRR), for solving Markov chains with a repeating structure. RRR uses ideas from renewal reward theory [160, Chapter 7] and leverages the repeating structure of the Markov chain to obtain closed-form expressions for metrics of interest such as T_{avg} and P_{avg}, or associated higher moments. *Importantly, our novel technique, RRR, provides the first closed-form analysis of multi-server systems with setup costs.* We now present the results of our RRR analysis of the M/M/k/setup. We defer the details of the RRR technique to Chapter 7.

Results of our analysis

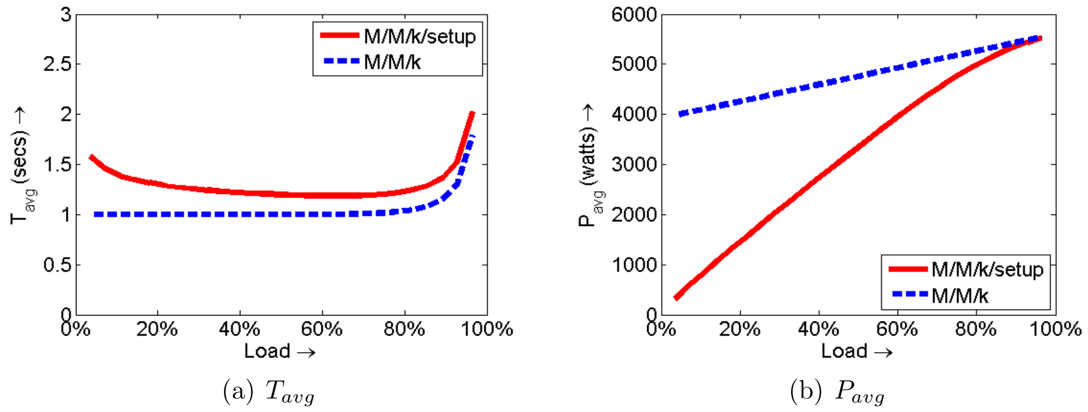
We use RRR to analyze setup times under various system parameters. In particular, we consider T_{avg} and P_{avg} for an M/M/k/setup. For comparison, we also consider these metrics for an M/M/k, where idle servers are not turned off. For all our results, we assume $k = 28$, an idle power consumption of 140W, and peak power consumption

(when the server is on or in setup) of 200W. These values are consistent with our implementation testbed in Section 2.3.2. In our results, we fix the mean request size ($\frac{1}{\mu}$) and analyze T_{avg} and P_{avg} as a function of load (see Definition 3.1). We vary load by varying the mean request rate as $\lambda = 28\mu \cdot \text{load}$ (see Equation (3.1)).

Figures 3.3, 3.4, and 3.5, show our results under mean setup times of $t_{setup} = 1$s, $t_{setup} = 10$s, and $t_{setup} = 100$s, respectively. We assume a mean request size of 1 second in all cases. Note that the M/M/k results (blue dashed lines) are the same for all cases since M/M/k is not affected by the setup time. For the M/M/k (blue dashed lines), we see that T_{avg} increases with load. This is to be expected. Likewise, P_{avg} increases with load. Note that P_{avg} increases linearly with load, as expected.

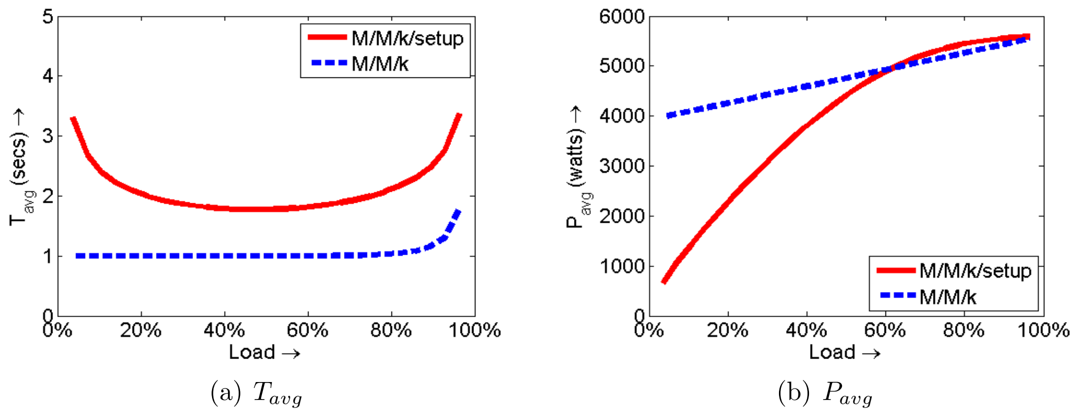
For the M/M/k/setup (red solid lines), we see an interesting trend. T_{avg} for the M/M/k/setup first decreases, and then increases. This can be explained as follows. When the system load is low, the mean time between incoming requests is high. Thus, when a new arrival comes into the system, there is some probability that it will find an empty system. In this case, the request will have to queue for the entire setup time to start execution. However, as the system load increases, the probability that an incoming request sees an empty system goes down. Thus, as load increases, the requests will not have to wait for the full setup time, because other servers will free up. Once the load gets really high, queueing time goes up again because of congestion. This explains the "bathtub" shaped T_{avg} curves. P_{avg} for the M/M/k/setup increases with load, as expected. However, when the setup time is really high (Figure 3.5(b)), P_{avg} plateaus quickly (to 5600W) as load increases because almost all 28 servers are either busy or in setup, thereby consuming 200W each.

In comparing M/M/k and M/M/k/setup, we see that M/M/k/setup is superior when t_{setup} is low, since P_{avg} is much lower under M/M/k/setup, and T_{avg} is similar for both policies. However, when t_{setup} is high, M/M/k has a much lower T_{avg}. This is because the short-term fluctuations in request rate (modeled by the Poisson process) are easily handled by the "always on" spare capacity of an M/M/k. However, these short-term fluctuations lead to servers being turned on and off under the M/M/k/setup, thereby incurring high setup costs. Surprisingly, when t_{setup} is high, M/M/k has a lower P_{avg} as well. The reason for this lower consumption is that under high setup times, the servers in M/M/k/setup are almost always either busy or in setup (owing to fluctuations in demand), thereby consuming peak power. Thus, M/M/k/setup is inferior when t_{setup} is high. This leads us to our first observation:



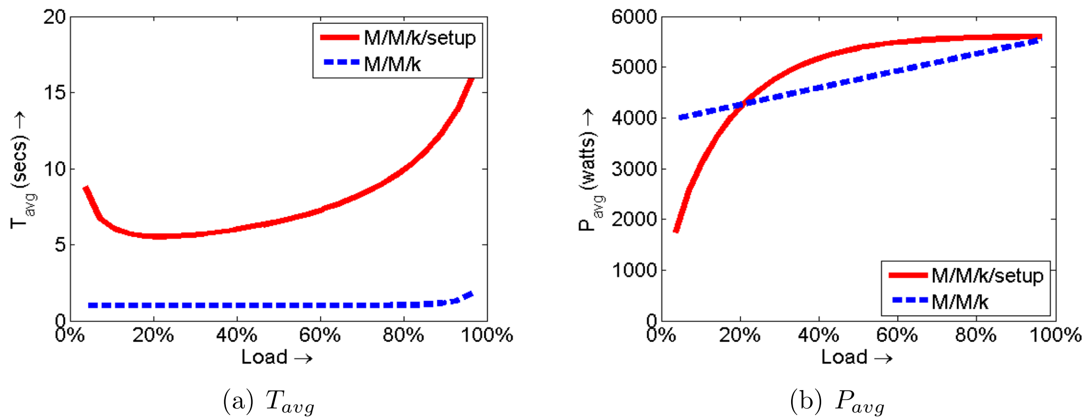
(a) T_{avg} (b) P_{avg}

Figure 3.3: Results for $t_{setup} = 1$s and mean request size of 1s.



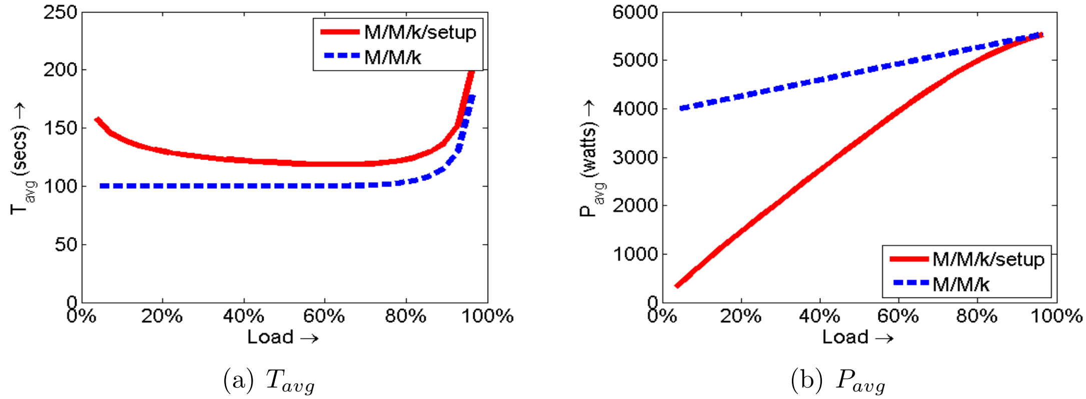
(a) T_{avg} (b) P_{avg}

Figure 3.4: Results for $t_{setup} = 10$s and mean request size of 1s.



(a) T_{avg} (b) P_{avg}

Figure 3.5: Results for $t_{setup} = 100$s and mean request size of 1s.



(a) T_{avg} (b) P_{avg}

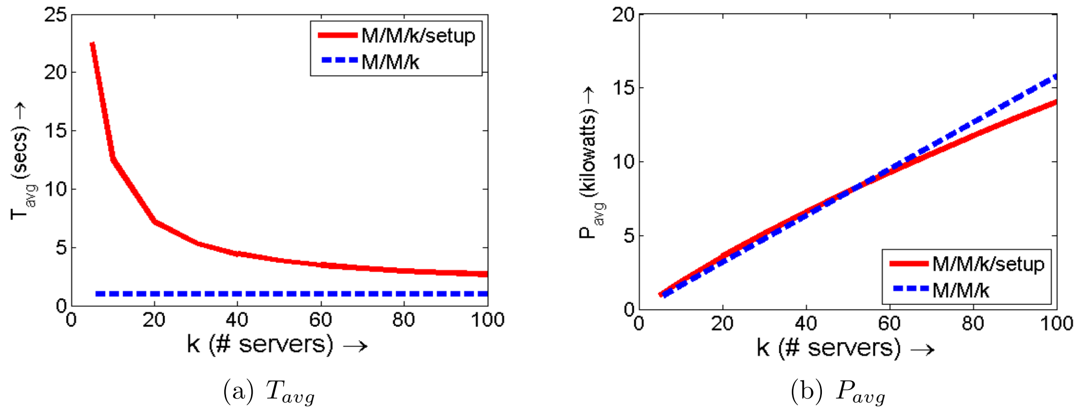
Figure 3.6: Results for $t_{setup} = 100$s and mean request size of 100s.

Observation 3.1 (The challenge presented by setup costs).
Setup costs incurred due to short-term fluctuations in demand negatively impact response time and power consumption, and thus should be avoided if possible, especially when the setup time is high.

This observation suggests that reactive approaches (see Section 2.2.2) will be hindered by short-term fluctuations.

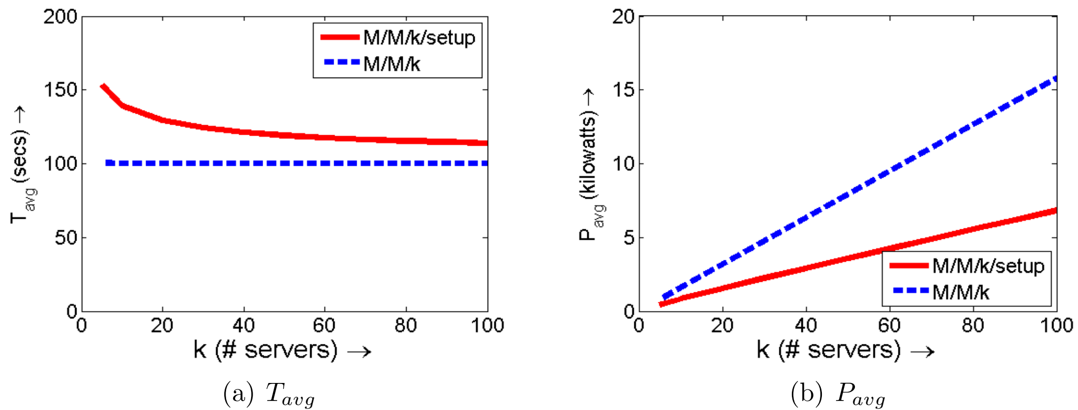
Figure 3.6 shows our results under a mean setup time of $t_{setup} = 100$s, but for a request size of 100s. We see that these results are qualitatively similar to Figure 3.3, where M/M/k/setup was superior. The reason why larger request sizes favor M/M/k/setup is because the queueing delay caused by setup times is not as severe when compared to the request size. Further, an increase in request size for a fixed load results in an increase in the inter-arrival time of requests. This leads to larger idle periods for servers, which in turn leads to greater power savings. Consequently, M/M/k/setup is superior to M/M/k when mean request size is high. *Thus, when comparing M/M/k/setup and M/M/k, what really matters is the ratio of setup time to request size. When this ratio is big, M/M/k is superior. When this ratio is small, M/M/k/setup is superior.* This leads us to our second observation:

Observation 3.2 (Factors affecting setup costs).
The adverse effects of setup costs increase in severity with the ratio of setup time to request size.



(a) T_{avg} (b) P_{avg}

Figure 3.7: Scaling results for $t_{setup} = 100$s and mean request size of 1s.



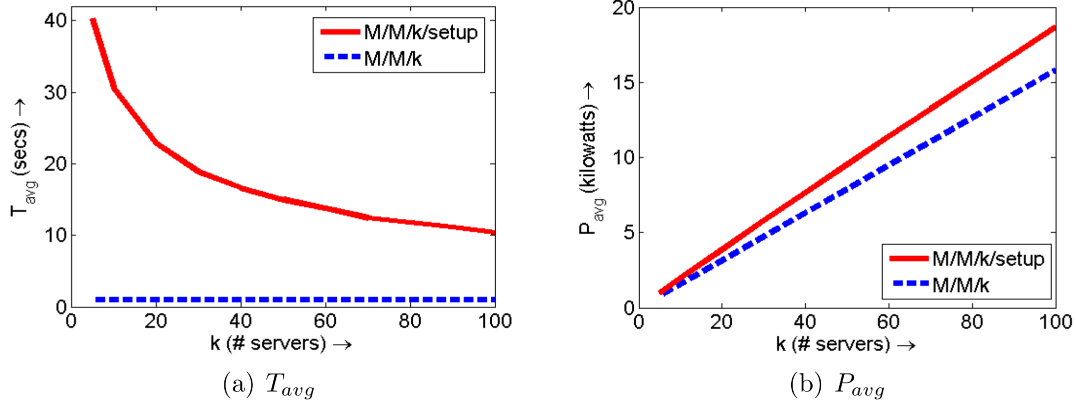
(a) T_{avg} (b) P_{avg}

Figure 3.8: Scaling results for $t_{setup} = 100$s and mean request size of 100s.

3.1.2 Effect of scale on setup costs

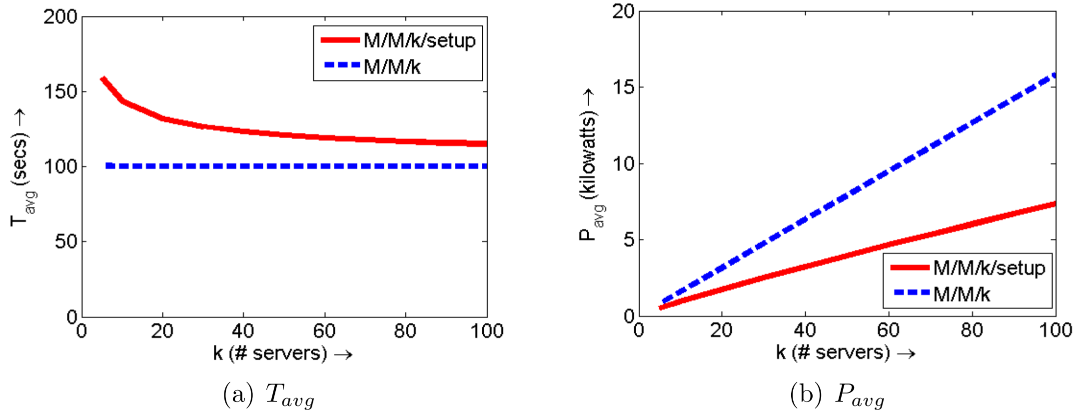
Thus far we have only considered a 28-server system in our analysis. We now study the effect of scale (number of servers, k) on setup costs. We fix the load at 30%, representing typical data center utilization [26]. We also fix t_{setup} at 100s.

Figures 3.7 and 3.8 show our results under mean request sizes of 1s and 100s respectively. In both cases, we see that the T_{avg} and P_{avg} for M/M/k/setup decrease as k increases. This is because as k increases, the probability that (any) one of the busy servers frees up soon, say in the next second, increases. Thus, an incoming request does not have to wait too long to begin execution, and consequently, fewer servers are in setup at any point of time. This result was also observed empirically by other researchers [198]. This leads us to our third observation on setup costs:



(a) T_{avg} (b) P_{avg}

Figure 3.9: Scaling results for t_{setup} = 100s and mean request size of 1s under deterministic setup times.



(a) T_{avg} (b) P_{avg}

Figure 3.10: Scaling results for t_{setup} = 100s and mean request size of 100s under deterministic setup times.

Observation 3.3 (Effect of scale on setup costs)**.**
The adverse effects of setup costs decrease in severity as the size of the data center (number of servers) scales up.

The analysis results above assumed exponential setup times. However, in real-world scenarios, setup times are typically deterministic. In order to analyze deterministic setup times, we resort to simulations. Figures 3.9 and 3.10 show our results under mean request sizes of 1s and 100s respectively, under deterministic setup times. We see that the relative ordering of the M/M/k and the M/M/k/setup policies and the trends in T_{avg} and P_{avg} do not change significantly when compared to the case of exponential setup times in Figures 3.7 and 3.8.

3.2 Uncertainty in Workload Demand

Section 3.1 above highlights the dangers posed by setup costs to data center power management. In particular, Observation 3.1 tells us that short-term fluctuations in demand can result in the incurrence of setup costs. When the setup time is high, the effects of setup costs can be even more severe. However, if fluctuations in demand can be predicted, we can avoid incurring setup costs by provisioning servers ahead of time. In this section we analyze the unpredictability in workload traces obtained from data center applications and from publicly available traces.

3.2.1 Analysis of workload traces

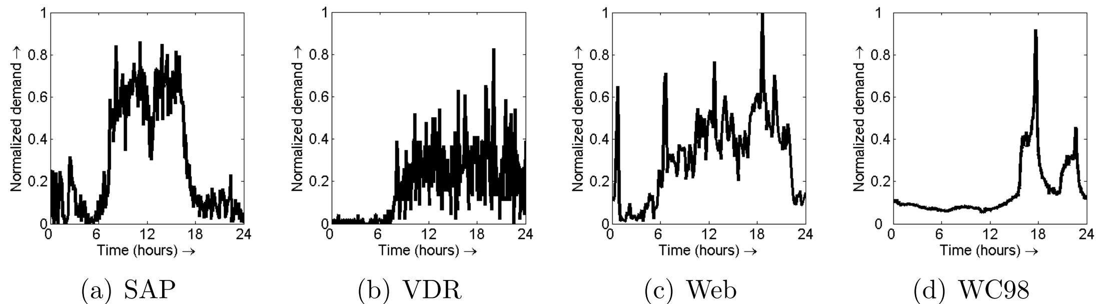
We analyze three application traces used in commercial data centers [60] and a publicly available web trace [95]. While some of the traces contain request rate data explicitly, others only contain aggregate server utilization data. For the purpose of analysis, we treat request rate data and utilization data as the same quantity – demand. We provide a brief description of these traces below before moving on to their analysis.

Description of traces

1. **SAP** is a five-week-long workload demand trace of an SAP enterprise application that was hosted in a commercial data center. The trace captures average CPU and memory usage as recorded every 5 minutes.
2. **VDR** is a ten-day-long trace containing request rate and system utilization data recorded every 5 minutes from a high-availability, multi-tier business-critical application serving customers from six continents.
3. **Web** is an eight-day-long trace of system utilization data from a popular Web service application with more than 85 million registered users in 22 countries, located in multiple data centers.
4. **WC98** is a three-month-long trace obtained from the Internet Traffic Archives that describes web requests received by the 1998 Soccer World Cup website.

Short-term fluctuations

Figure 3.11 shows a typical 24-hour workload demand for the SAP, VDR, Web, and WC98 traces. The demand has been normalized by the maximum demand in



(a) SAP (b) VDR (c) Web (d) WC98

Figure 3.11: Demand for a single day for (a) the SAP trace, (b) the VDR trace, (c) the Web trace, and (d) the WC98 trace.

each trace. We see that the demand varies a lot within a 24-hour interval. For example, the SAP trace demand varies from a minimum of almost 0 to a maximum of approximately 0.8, which corresponds to roughly 80% CPU utilization based on the actual trace. From Figure 3.11, we also see that there are a lot of short-term fluctuations in demand. This leads us to the following observation:

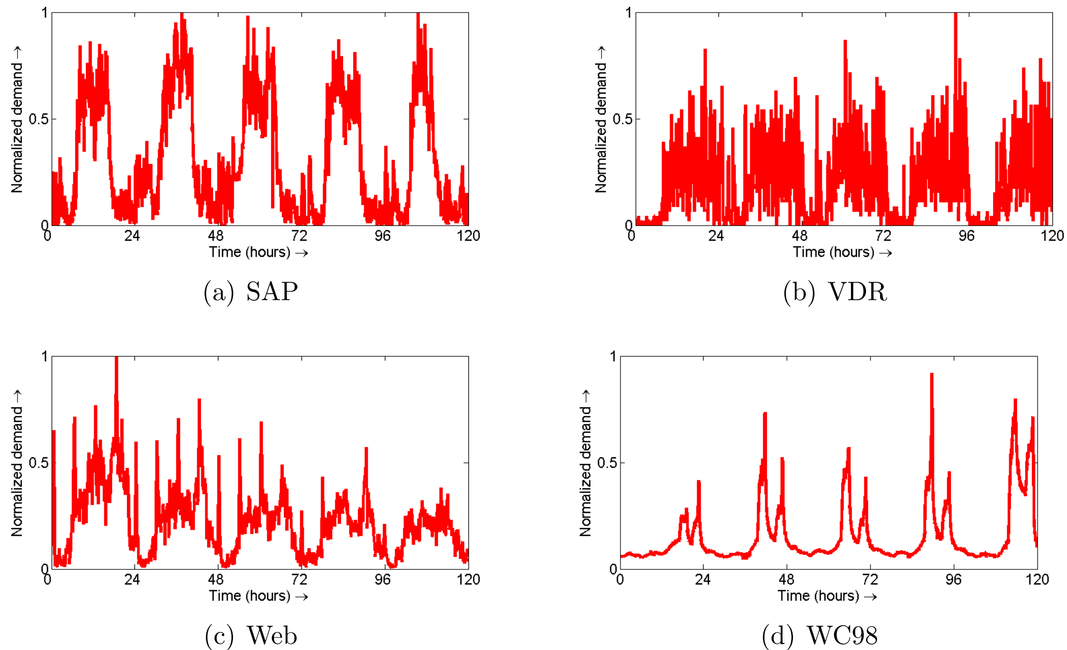
Observation 3.4 (Variance in workload demand).
Workload demand is often bursty in nature due to short-term fluctuations.

Long-term patterns

Figure 3.12 shows a 5-day time-series of the SAP, VDR, Web, and WC98 traces. We see that the demand for each trace roughly repeats on a daily basis. To further investigate this periodicity, we use Fast Fourier Transform to find the periodogram of the time-series data [32]. The periodogram reveals a peak at 24 hours, indicating that the traces have a strong daily pattern (period of 24 hours). The periodogram does not reveal any other significant peaks. Based on this analysis, we conclude that:

Observation 3.5 (Predictability of workload demand).
While there is large variability in workload demands, most workloads exhibit predictable long-term (daily) patterns. However, short-term fluctuations cannot be predicted based on periodicity.

This observation suggests that purely predictive approaches (see Section 2.2.1) might be insufficient for handling data center workloads. For a complete analysis of the above four traces, we refer the reader to our papers [60, 61].



(a) SAP (b) VDR

(c) Web (d) WC98

Figure 3.12: 5-day time-series demand plots for (a) the SAP trace, (b) the VDR trace, (c) the Web trace, and (d) the WC98 trace.

Consequences of short-term fluctuations

Based on Observation 3.5, we deduce that data center workload demands often consist of a predictable long-term daily pattern and unpredictable short-term fluctuations. The long-term patterns in demand can be handled by proactively provisioning servers for the predicted demand. In order to verify this hypothesis, we implement a simple single-tier CPU-bound application on a 10-server system, and replay the above four traces using a predictive controller (see [60, 61] for more details). We find that the purely predictive controller results in about 6% – 19% SLA violations across all four traces. This suggests that the predictable daily pattern accounts for around 80% of the total demand, meaning that short-term fluctuations make up less than 20% of the total demand. However, the 6% – 19% range of SLA violations suggest that purely predictive policies will not be able to meet T_{95} SLAs which require less than 5% SLA violations. To account for short-term fluctuations, we also experiment with a hybrid approach by adding a reactive controller to the above predictive controller. Our hybrid approach results in 5% – 11% SLA violations. While this is an improvement over predictive, the percentage of SLA violations is still quite high.

This is to be expected given Observation 3.1 and the fact that short-term fluctuations account for a non-trivial portion of the total demand. We also experiment with a purely reactive approach, but this results in significant SLA violations, as should be expected. Based on these results, we make the following observation:

Observation 3.6 (The challenge presented by short-term fluctuations)**.**
While short-term fluctuations often account for only a fraction of the total workload demand, they can lead to significant SLA violations, even when using hybrid (reactive-predictive) provisioning approaches.

Given this observation, it seems that the only solution to handling unpredictable short-term fluctuations is to over-provision capacity. In Chapter 4, we present a new approach, *AutoScale*, to handling short-term fluctuations in demand in the presence of setup times.

3.2.2 Load spikes

Another challenge in dealing with unpredictable workload demand is the possibility of load spikes. While load spikes, or abrupt changes in load, are not necessarily a daily occurrence in data centers, several instances of load spikes have been documented for web workloads. Important events, such as the September 11 attacks [113, 88], earthquakes or other natural disasters [186], slashdot effects [7], Black Friday shopping [44], or sporting events, such as the Super Bowl [145] or the Soccer World Cup [20], are common causes of load spikes for website traffic. Service outages [147] or server failures [162] can also result in abrupt changes in load caused by a sharp drop in capacity. While some of the above events are predictable, most of them *cannot* be predicted in advance.

Load spikes are especially problematic since adding capacity requires some setup time. Even if we instantaneously detect a spike in load, it will still take the system at least the setup time to add the required capacity. In our lab, the setup time for turning on an additional server is approximately 4-5 minutes. Similar setup times have also been reported in recent literature [93, 129]. Likewise, the setup time needed to create virtual machines (VMs) can range anywhere from 30 seconds – 1 minute if the VMs are locally created (based on our measurements using kvm [102]) or 2 – 10 minutes if the VMs are obtained from a cloud computing platform (see, for example, [13, 108]). All these numbers are extremely high, and can result in significant SLA violations.

Analysis of load spikes

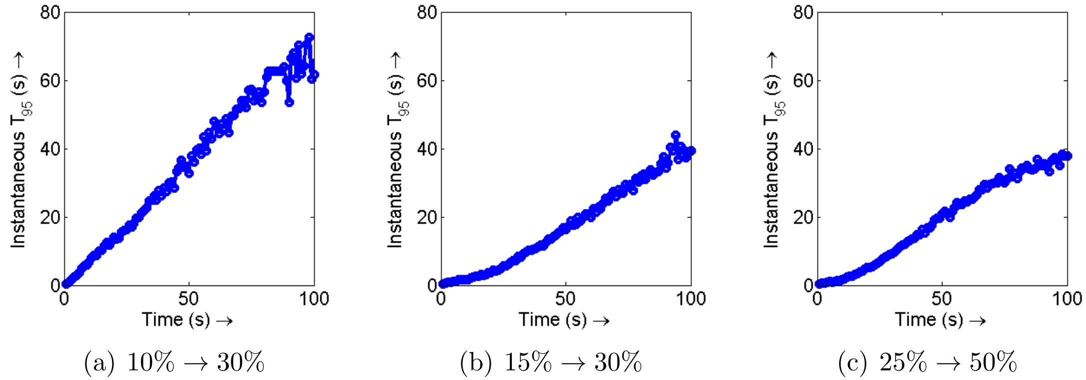
In order to analyze load spikes, we use our implementation testbed described in Section 2.3.2. We use synthetic demand traces to experimentally study the effects of load spikes on instantaneous T_{95} (see Definition 2.3). We choose instantaneous T_{95} over T_{95} to highlight the dynamic effect of load spikes on response time. We normalize all traces and report the request rate as a percentage of the peak request rate our testbed can handle. In our experiments, we initially load the system with some x% request rate. Then, at some time t, we instantaneously increase the request rate to y%, where $x < y \leq 100$. We optimistically assume that the system immediately detects this load spike and provisions the required number of servers to handle the new y% request rate. However, this additional capacity only comes online after the setup time, at $t + t_{setup}$. During t to $t + t_{setup}$, the system is in overload.

Figure 3.13 shows our experimental results under load spikes of (a) 10% → 30%, (b) 15% → 30%, and (c) 25% → 50%. Here, the setup time is $t_{setup} = 5$ mins. In these experiments, the load spike occurs at the beginning of the experiment, and lasts for the duration of the experiment. In all cases, we see that the instantaneous T_{95} rises very steeply. The rise is steeper in Figure 3.13(a) than in Figure 3.13(b), though in both cases the load spikes to 30%. Also, the rise in instantaneous T_{95} is very similar in Figures 3.13(b) and 3.13(c), where load doubles. These results suggest that the rise in instantaneous T_{95} depends on the size of the load jump relative to the initial load.

Figure 3.14 shows our experimental results under a 15% → 30% load spike for (a) $t_{setup} = 50$s, (b) $t_{setup} = 20$s, and (c) $t_{setup} = 5$s. Note the difference in the axes for each figure. In these experiments, the load spike occurs at the 10s mark and lasts for the duration of the experiment. Once the setup time is complete, at $10 + t_{setup}$, the required capacity is bought online. Interestingly, the instantaneous T_{95} does not immediately drop at $10 + t_{setup}$ when the required capacity is online. This is because of the backlog in requests created during the overload period. We see that the length of the setup time impacts the duration over which instantaneous T_{95} rises, as expected. Even for a setup time of 5s, we see that instantaneous T_{95} rises very quickly from 500ms to 1800ms.

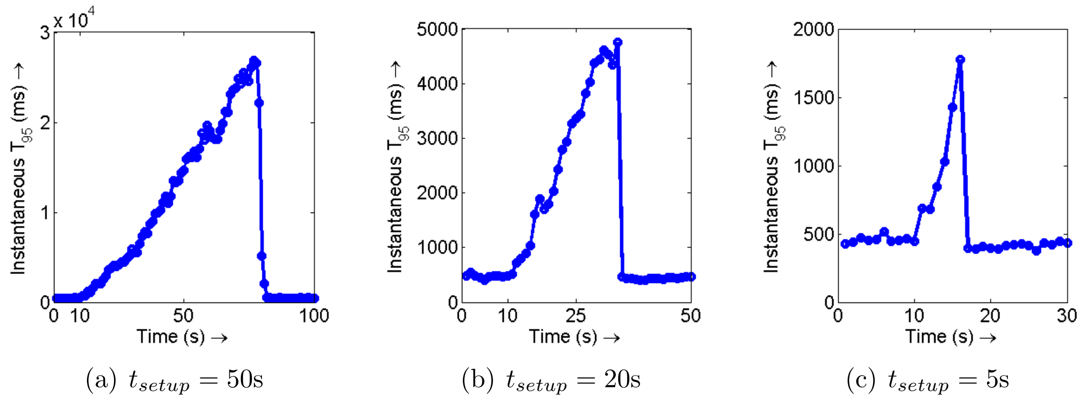
A more complete evaluation, including results for load spikes seen in real traces, is presented in Chapter 5. We conclude this section by summarizing our above analysis:

Observation 3.7 (The challenge presented by load spikes)**.**
Load spikes, though rare in occurrence, can lead to a steep rise in response time, even if the setup time is only 5 seconds.



(a) 10% → 30% (b) 15% → 30% (c) 25% → 50%

Figure 3.13: Experimental results for instantaneous T_{95} under a load spike of (a) 10% → 30%, (b) 15% → 30%, and (c) 25% → 50%. Here, the setup time is t_{setup} = 5 mins. The load spike occurs at the beginning of the experiment and lasts for the duration of the experiment.



(a) t_{setup} = 50s (b) t_{setup} = 20s (c) t_{setup} = 5s

Figure 3.14: Experimental results for instantaneous T_{95} under a 15% → 30% load spike for (a) t_{setup} = 50s, (b) t_{setup} = 20s, and (c) t_{setup} = 5s. The load spike occurs at the 10s mark and lasts for the duration of the experiment.

Given this observation, it seems that the only solution to handling load spikes is to over-provision capacity or resort to corrective measures such as admission control. In Chapter 5, we present a new approach, *SoftScale*, to handling load spikes in multi-tier data centers.

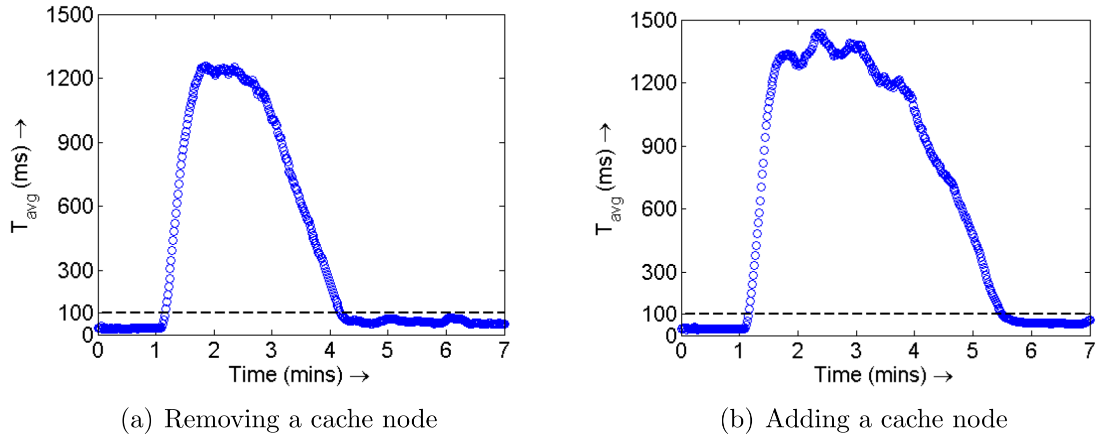
3.3 Stateful Servers

Stateful servers, such as caching tier servers, are typically [128] provisioned with massive amounts of expensive and power-hungry DRAM. Thus, dynamic provisioning for the caching tier can lead to huge savings in cost and power. The big challenge in dynamically provisioning stateful servers is the possibility of data unavailability when changing the total cache capacity of the system. For example, if one cache server is taken offline to reduce power consumption, the data stored on this server now becomes unavailable to the system, leading to a lower cache hit rate. This results in a rise in response time as requests now go to the slower disk for data fetches. However, over time, the hit rate of the system improves as the hot data items that were originally stored on the now-offline server get cached on the remaining (online) cache servers. Thus, response time suffers, but only for a finite period of time.

3.3.1 Consequences of dynamically changing the cache size

In order to study the consequences of dynamically changing the caching tier size, we use our implementation testbed described in Section 2.3.2, with a few modifications. The caching tier comprises 20 nodes running memcached [59], each with up to 10GB of memory for caching, hosted on 5 physical servers (Xeon X5650). The data tier comprises a server (Xeon E5520) with 5 disks running a BerkeleyDB [146] database with a billion key-value pairs (250GB). Our workload request is a PHP script that runs on the application server, and consists of 20 independent key-value GET requests. The requested keys follow a Zipf [143] distribution. Each of the 20 GET requests either hits in the memcached, or, if it misses, goes to the database. We pessimistically force cache misses to result in database disk accesses by avoiding the database page cache. If a GET request hits in the memcached, its response time is $T_{mem} = 0.3ms$, which is the time to retrieve a key-value pair from memcached. If a GET request goes to the database, its response time is on the order of 8ms. The actual response time depends on the contention at the database. In this set of experiments, we require that the T_{avg} for the entire request (collection of 20 GET requests) should be no more than 100ms. That is, $T_{avg} \leq$ 100ms.

Figure 3.15(a) shows our experimental results where we scale down from 4 to 3 cache nodes at the 1 minute mark because the request rate is low enough to require only 3 nodes worth of cached data. We see that removing a cache node at the 1 minute mark leads to a steep increase in T_{avg}. As requests continue to arrive at the 3-node cache tier, the hot data items that were originally stored on the node



(a) Removing a cache node

(b) Adding a cache node

Figure 3.15: Experimental results for the case of (a) removing and (b) adding a caching tier node. The change in capacity is made at the 1 minute mark. The black dashed line represents our 100ms SLA.

that was removed get cached on the remaining three online nodes. This results in improving the hit rate of the caching tier. At around the 4 minute mark, T_{avg} goes below the SLA. Thus, removing the cache node resulted in a 3 minute interval of SLA violations.

SLA violations can also occur when we add a cache node to the caching tier. This is because the newly added node is initially "cold" and all requests to that node result in misses. Figure 3.15(b) shows our experimental results where we scale up from 3 to 4 cache nodes at the 1 minute mark. Again, we see that adding a cache node at the 1 minute mark leads to a steep increase in T_{avg}. Over time, T_{avg} goes back to normal (below the SLA) as the newly added cache node gets warmed up. In this case, adding the cache node resulted in a 4 minute interval of SLA violations. A more complete evaluation of dynamically changing the cache size, along with details of our experiments, is presented in Chapter 6. We now conclude this section by summarizing our above experimental results:

Observation 3.8 (The challenge presented by stateful servers)**.**
Adding or removing a caching tier server can lead to (temporary) unavailability of certain data items, resulting in a steep rise in response times.

The above observation is probably the reason why there has been no prior work on dynamic provisioning of the caching tier. In Chapter 6, we present a new approach, *CacheScale*, to dynamically provision the caching tier.

3.4 Chapter Summary

In this chapter we analyzed the challenges faced by dynamic server provisioning. First, we examined setup costs, which are incurred due to dynamic addition and removal of servers in response to fluctuations in demand. We presented the first analysis of setup costs, and found that they can lead to a significant increase in response time *and* power consumption (see Observation 3.1). Then, we analyzed workload demand traces from real-world commercial and web applications, and found that there is often some degree of unpredictability in demand (see Observation 3.6). Most of this unpredictability can be attributed to short-term fluctuations in demand, which result in the incurrence of setup costs. In some cases, the unpredictability is due to load spikes, which lead to an abrupt rise in response times (see Observation 3.7). Finally, we experimentally analyzed dynamic provisioning of the caching tier, and found that this can lead to temporary unavailability of cached data, resulting in a steep rise in response times (see Observation 3.8). The next three chapters provide practical dynamic provisioning solutions that overcome the above-mentioned challenges.

Chapter 4

AutoScale: Robust Dynamic Server Provisioning

In this chapter we address the challenges presented by unpredictable fluctuations in demand and setup costs. Fluctuations in demand necessitate dynamic addition and removal of servers, leading to the incurrence of setup costs (see Observation 3.6). Setup costs, in turn, severely impact response time and power consumption (see Observation 3.1), as illustrated by our analytical results in Figures 3.3, 3.4 and 3.5.

We present *AutoScale*, a practical dynamic server provisioning solution that overcomes these challenges. *AutoScale* greatly reduces the number of servers needed in data centers driven by unpredictable load, while meeting response time SLAs. *AutoScale* has two key features:
(i) it automatically maintains just the right amount of spare capacity to handle short-term fluctuations in request rate; and
(ii) it is *robust* not just to unpredictable changes in request rate, but also unpredictable changes in *request size* and *server efficiency*.

We introduce the problem and discuss the scope of this chapter in Section 4.1. We then briefly describe our experimental setup for this chapter in Section 4.2. In Section 4.3, we first experimentally evaluate existing server provisioning policies, and then introduce our robust dynamic server provisioning policy, *AutoScale*. We investigate the applicability of *AutoScale* under sleep states in Section 4.5, and under low-power idle states in Section 4.6. We then evaluate the robustness properties of *AutoScale* in Section 4.7 by considering unpredictable changes in request size and server speeds. We discuss prior work in Section 4.8 and conclude with a summary of this chapter in Section 4.9.

4.1 Introduction

Many networked services, such as Facebook and Amazon, are provided by multi-tier data center infrastructures. A primary goal for these applications is to provide good response time to users; these response time targets typically translate to some response time SLAs (see Definition 1.2). In an effort to meet these SLAs, data center operators typically over-provision the number of servers to meet their estimate of peak load. These servers are left "always on," leading to only 10–30% server utilization [21, 26]. In fact, [167] reports that the average data center server utilization is only 18% despite years of deploying virtualization. Low utilization is problematic because servers that are on, while idle, still utilize 60% or more of peak power.

To reduce wasted power, we consider intelligent dynamic server provisioning, which aims to match the number of active servers with the current load, in situations where future load is unpredictable. Servers which become idle when load is low could be either *turned off*, saving power, or *loaned out* to some other application, or simply *released* to a cloud computing platform, thus saving money. Fortunately, the bulk of the servers in a multi-tier data center are application servers, which are *stateless*, and are thus easy to turn off or give away – for example, one reported ratio of application servers to data servers is 5:1 [55]. We therefore focus our attention in this chapter on dynamic server provisioning of these front-end application servers. We consider dynamic server provisioning of stateful caching servers in Chapter 6.

Part of what makes dynamic server provisioning difficult is the *setup cost* of getting servers back on/ready. For example, in our lab the setup time for turning on an application server is 260 seconds, during which time power is consumed at the peak rate of 200W. Likewise, recent literature reported the setup time of server-class machines to be on the order of minutes [93, 129]. Sadly, little has been done to reduce the setup overhead for servers. In particular, sleep states, which are prevalent in mobile devices, have been very slow to enter the server market. Even if future hardware reduces the setup time, there may still be software imposed setup times such as installing software updates which occurred when the server was unavailable [55] or performing membership updates in distributed systems [129]. Likewise, the setup cost needed to create virtual machines (VMs) can range anywhere from 30s – 1 minute if the VMs are locally created (based on our measurements using kvm [102]) or 2 – 10 minutes if the VMs are obtained from a cloud computing platform (see, for example, [13, 108]). All these numbers are extremely high when compared to the typical response time SLA of half a second [48]. For an in-depth analysis of the consequences of setup costs, we refer the reader to Section 3.1.

The goal of dynamic server provisioning is to scale capacity with unpredictably changing load in the face of high setup costs. While there has been much prior work on this problem, all of it has only focussed on one aspect of changes in load, namely, fluctuations in request rate. This is already a difficult problem, given high setup costs, and has resulted in many policies, including reactive approaches [114, 141, 57, 190, 196, 53] that aim to react to the current request rate, predictive approaches [107, 152, 36, 86] that aim to predict the future request rate, and mixed reactive-predictive approaches [40, 41, 28, 174, 76, 173, 60, 61]. However, in reality there are many other ways in which load can change. For example, *request size* (work associated with each request) can change, if new features or security checks are added to the application. As a second example, *server efficiency* can change, if any abnormalities occur in the system, such as internal service disruptions, slow networks, or maintenance cycles. These other types of load fluctuations are all too common in data centers, and have not been addressed by prior work in dynamic server provisioning.

We propose a *new approach to dynamic server provisioning*, which we call *AutoScale*. To describe *AutoScale*, we decompose it into two parts: *AutoScale*-- (see Section 4.3.5), which is a precursor to *AutoScale* and handles only the narrower case of unpredictable changes in request rate, and the full *AutoScale* policy (see Section 4.7.3), which builds upon *AutoScale*-- to handle all forms of changes in load.

While *AutoScale*-- addresses a problem that many others have looked at, it does so in a very different way. While prior approaches aim at predicting the future request rate and scaling *up* the number of servers to meet this predicted rate, which is clearly difficult to do when request rate is, by definition, unpredictable, *AutoScale*-- does not attempt to predict future request rate. Instead, *AutoScale*-- demonstrates that it is possible to achieve SLAs for real-world workloads by simply being conservative in scaling *down* the number of servers: not turning servers off recklessly. One might think that this same effect could be achieved by leaving a fixed buffer of, say, 20% extra servers on at all times. However, the extra capacity (20% in the above example) should *change* depending on the current load. *AutoScale*-- does just this – it automatically maintains just the right number of servers in the on state at every point in time. This results in much lower power/resource consumption. In Section 4.3.5, we evaluate *AutoScale*-- on a suite of six different real-world traces, comparing it against five different server provisioning policies commonly used in the literature. We demonstrate that in all cases, *AutoScale*-- significantly outperforms other policies, meeting response time SLAs while greatly reducing the number of servers needed, as shown in Table 4.2.

To fully investigate the applicability of *AutoScale*--, we experiment with multiple sleep states that have setup times ranging from 260 seconds all the way down to 20 seconds and sleep power ranging from 140 Watts to 0 Watts in Section 4.5. We also experiment with multiple server idle power consumption values ranging from 140 Watts all the way down to 0 Watts in Section 4.6. Our results indicate that *AutoScale*-- can provide significant benefits across the entire spectrum of sleep states and idle power, as shown in Figures 4.9, 4.10 and 4.13. Our investigation of sleep states also helps us determine which sleep states are useful for dynamic server provision. We present this analysis in Section 4.5.3.

To handle a broader spectrum of possible changes in load, including unpredictable changes in the request size and server efficiency, we introduce the *AutoScale* policy in Section 4.7.3. While prior approaches to dynamic server provisioning for multi-tier applications react only to changes in the request rate, *AutoScale* uses a novel capacity inference algorithm, which allows it to determine the appropriate capacity regardless of the source of the change in load. Importantly, *AutoScale* achieves this *without* requiring any knowledge of the request rate or the request size or the server efficiency, as shown in Tables 4.4, 4.5 and 4.6.

To evaluate the effectiveness of *AutoScale*, we build a three-tier testbed. While our implementation involves physically turning servers on and off, one could instead imagine that idle servers are "given away," and there is a setup time to get the server back. We evaluate all policies on three metrics: T_{95}, the 95th percentile of response time (see Definition 2.2), which represents our SLA; P_{avg}, the average power usage (see Definition 2.4); and N_{avg}, the average capacity, or number of servers in use (including those idle and in setup, see Definition 2.5). Our goal is to meet the response time SLA, while keeping P_{avg} and N_{avg} as low as possible. The drop in P_{avg} represents the possible savings in power obtained by turning servers off, while the drop in N_{avg} represents the potential capacity/servers available to be given away to other applications or to be released back to the cloud so as to save on rental costs.

This chapter makes the following contributions:

- We overturn the common wisdom that dynamic server provisioning requires "knowing the future load and planning for it," which is at the heart of existing predictive provisioning policies. Such predictions are simply not possible when workloads are unpredictable, and, we furthermore show they are unnecessary, at least for the range of variability in our workloads. We demonstrate that simply provisioning carefully and not turning servers off recklessly achieves better performance than existing policies that are based on predicting current load or over-provisioning to account for possible future load.

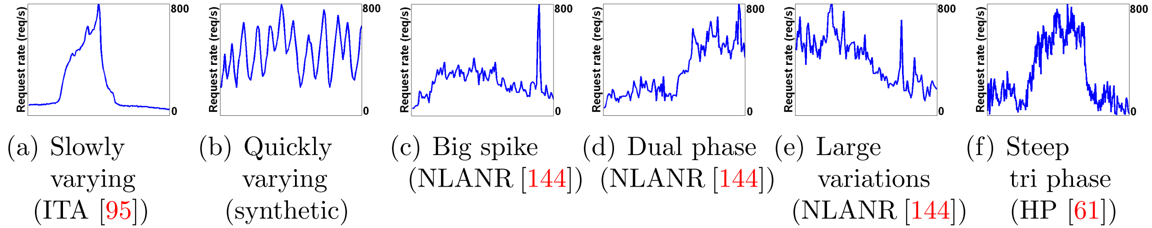
- We introduce our capacity inference algorithm which allows us to determine the appropriate capacity at any point of time in response to changes in request rate, request size and/or server efficiency, without any knowledge of these quantities (see Section 4.7.3). We demonstrate that *AutoScale*, via the capacity inference algorithm, is robust to all forms of changes in load, including unpredictable changes in request size and unpredictable degradations in server speeds, within the range of our traces. In fact, for our traces, *AutoScale* is robust to even a 4-fold increase in request size. To the best of our knowledge, *AutoScale* is the first policy to exhibit these forms of robustness for multi-tier applications. As shown in Tables 4.4, 4.5 and 4.6, other policies are simply not comparable on this front.
- We provide a sensitivity analysis of sleep states for different server provisioning policies, including *AutoScale*, and determine the regime of sleep states that would be advantageous for power management (see Section 4.5). We also provide a sensitivity analysis of lower idle power for different server provisioning policies (see Section 4.6).

4.2 Experimental Setup

4.2.1 Our experimental testbed

We use our implementation testbed described in Section 2.3.2 and illustrated in Figure 2.1 for all the experiments in this chapter. The key-value based workload that we use for our experiments is also described in Section 2.3.2. Each workload request in our experiments corresponds to an average of roughly 3,000 key-value fetches, which translates to a mean request size of approximately 120 ms, assuming no resource contention. The request size distribution is highly variable, with the largest request being roughly 20 times the size of the smallest request. We use the Zipf [143] distribution to model the popularity of requests. To minimize the effects of cache misses in the memcached layer (which could result in an unpredictable fraction of the requests violating the T_{95} SLA), we tune the parameters of the Zipf distribution so that only a negligible fraction of requests miss in the memcached layer.

In this chapter we employ server provisioning on the stateless application servers only, as they maintain no volatile state. Stateless servers are common in today's application platforms, such as those used by Facebook [55], Amazon [48] and Windows Live Messenger [40]. We consider provisioning of stateful tier servers in Chapter 6.



(a) Slowly varying (ITA [95]) (b) Quickly varying (synthetic) (c) Big spike (NLANR [144]) (d) Dual phase (NLANR [144]) (e) Large variations (NLANR [144]) (f) Steep tri phase (HP [61])

Figure 4.1: Description of the traces we use for our experiments.

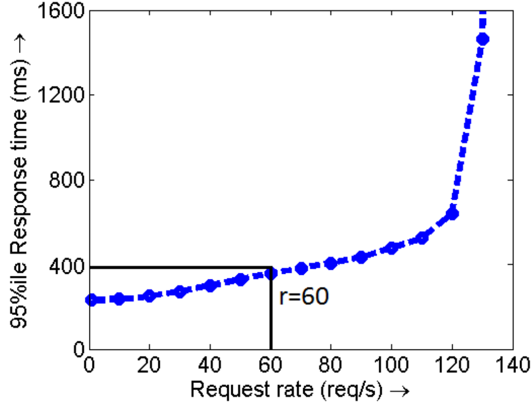
4.2.2 Trace-based arrivals

We use a variety of arrival traces to generate the request rate in our experiments, most of which are drawn from real-world traces. Figure 4.1 describes these traces. In our implementation testbed, the seven memcached servers can together handle at most 800 requests per second, which corresponds to roughly 300,000 key-value fetches per second at each memcached server. Thus, we scale the arrival traces such that the maximum request rate into the system is 800 req/s. Further, we scale the duration of the traces to 2 hours. We evaluate our policies against the full set of traces (see Table 4.2 for results).

4.3 Evaluation I: Fluctuations in Request Rate

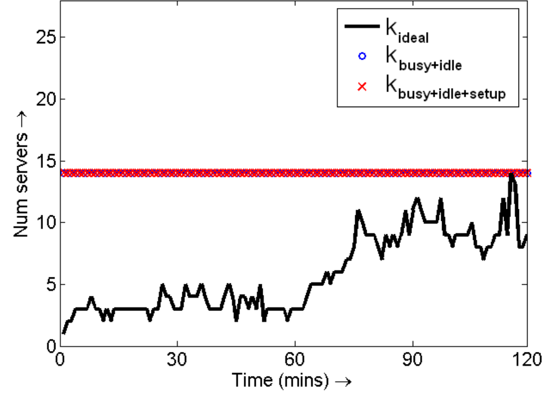
This section and the next both involve implementation and performance evaluation of a range of server provisioning policies. Each policy will be evaluated against the six traces described in Figure 4.1. We will present detailed results for the Dual phase trace and show summary results for all traces in Table 4.2. The Dual phase trace is chosen because it is quite bursty and also represents the diurnal nature of typical data center traffic, whereby the request rate is low for a part of the day (usually the night time) and is high for the rest (day time). The goal throughout will be to meet 95%ile guarantees of $T_{95} = 400 - 500$ ms, while minimizing the average power consumed by the application servers, P_{avg}, or the average number of application servers used, N_{avg}. Note that P_{avg} largely scales with N_{avg}.

It would be equally easy to use 90%ile or 99%ile response time guarantees. Likewise, we could easily have aimed for 300ms or 1 second response times rather than 500ms. Our choice of SLA is motivated by recent studies [173, 107, 132, 48] which indicate that 95%ile guarantees of hundreds of milliseconds are typical.



95%ile response time vs. request rate

Figure 4.2: A single server can optimally handle 60 req/s.



T_{95}=291ms, P_{avg}=2,323W, N_{avg}=14

Figure 4.3: *AlwaysOn*.

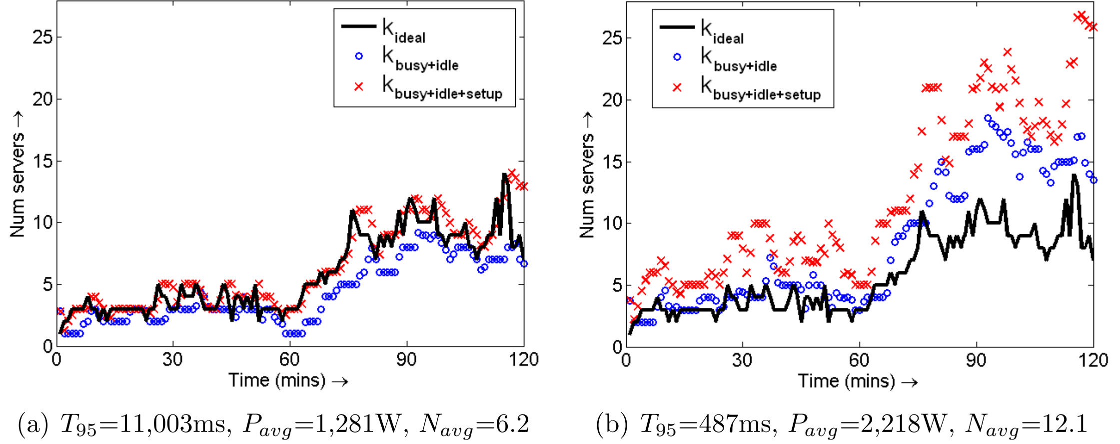
For server provisioning, we want to choose the number of servers at time t, $k(t)$, such that we meet a 95%ile response time goal of $400 - 500$ ms. Figure 4.2 shows measured 95%ile response time at a *single server* versus request rate. According to this figure, for example, to meet a 95%ile goal of 400 ms, we require the request rate to a single server to be no more than $r = 60$ req/s. Hence, if the total request rate into the data center at some time t is say, $R(t) = 300$ req/s, we know that we need at least $k = \lceil 300/r \rceil = 5$ servers to ensure our 95%ile SLA.

4.3.1 AlwaysOn

The *AlwaysOn* policy [183, 40, 86] is important because this is what is currently deployed by most of the industry. The policy selects a fixed number of servers, k, to handle the peak request rate and always leaves those servers on. In our case, to meet the 95%ile SLA of 400ms, we set $k = \lceil R_{peak}/60 \rceil$, where $R_{peak} = 800$ req/s denotes the peak request rate into the system. Thus, k is fixed at $\lceil 800/60 \rceil = 14$.

Realistically, one does not know R_{peak}, and it is common to overestimate R_{peak} by a factor of 2 (see, for example, [107]). In this chapter we empower *AlwaysOn*, by assuming that R_{peak} is known in advance.

Figure 4.3 shows the performance of *AlwaysOn*. The solid line shows k_{ideal}, the *ideal* number of servers/capacity which should be on at any given time, as given by $k(t) = \lceil R(t)/60 \rceil$. Circles are used to show $k_{busy+idle}$, the number of servers which are actually on, and crosses show $k_{busy+idle+setup}$, the actual number of servers that



(a) T_{95}=11,003ms, P_{avg}=1,281W, N_{avg}=6.2 (b) T_{95}=487ms, P_{avg}=2,218W, N_{avg}=12.1

Figure 4.4: (a) *Reactive* and (b) *Reactive* with extra capacity.

are on or in setup. For *AlwaysOn*, the circles and crosses lie on top of each other since servers are never in setup. Since most of the time *AlwaysOn* is clearly over-provisioning, T_{95} is well below the desired SLA. However, N_{avg} and P_{avg} are quite high, with $N_{avg} = \lceil \frac{800}{60} \rceil = 14$ and $P_{avg} = 2323W$, and similar values for the other traces (see Table 4.2).

4.3.2 Reactive

The *Reactive* policy (see, for example, [173]) reacts to the current request rate, attempting to keep exactly $\lceil R(t)/60 \rceil$ servers on at time t, in accordance with the solid line. However, because of the setup time of 260s, *Reactive* lags in turning servers on. In our implementation of *Reactive*, we sample the request rate every 20 seconds, adjusting the number of servers as needed.

Figure 4.4(a) shows the performance of *Reactive*. By reacting to the current request rate and adjusting the capacity accordingly, *Reactive* is able to bring down P_{avg} and N_{avg} by as much as a factor of two or more, when compared with *AlwaysOn*. This is a huge win. Unfortunately, the response time SLA is almost never met and is typically exceeded by a factor of at least 10-20 (as in Figure 4.4(a)), or even by a factor of 100 (see Table 4.2). *Reactive* suffers due to the setup lag, resulting in servers not being on when needed, providing insufficient capacity for the incoming requests. Also, because of the bursty nature of the trace, a drop in request rate is often closely followed by a rise in request rate. But because *Reactive* scales down capacity in response to a drop in request rate, there is insufficient capacity when the request rate increases again, and this hurts response times.

4.3.3 Reactive with extra capacity

One might think that the response times under *Reactive* would improve a lot by just adding some $x\%$ extra capacity at all times. This $x\%$ extra capacity can be achieved by running *Reactive* with a different r setting. Unfortunately, for this trace, it turns out that to bring T_{95} down to our desired SLA, we need 100% extra capacity at all times, which corresponds to setting $r = 30$. This brings T_{95} down to 487 ms, but causes power to jump up to the levels of *AlwaysOn*, as illustrated in Figure 4.4(b). It is even more problematic that each of our six traces in Figure 4.1 requires a different $x\%$ extra capacity to achieve the desired SLA (with $x\%$ typically ranging from 50% to 200%), rendering such a policy impractical.

4.3.4 Predictive

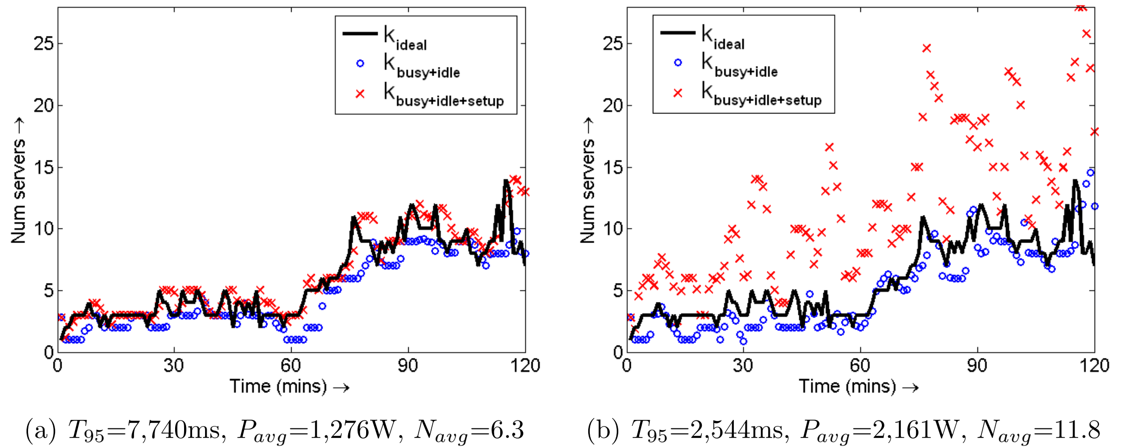
Predictive policies attempt to predict the request rate 260 seconds from now. This section describes two policies that were used in many papers [30, 81, 150, 183] and were found to be the most powerful by [107].

Predictive - Moving Window Average (MWA)

In the *MWA* policy, we consider a "window" of some duration (say, 10 seconds). We average the request rates during that window to deduce the predicted rate during the 11th second. Then we slide the window to include seconds 2 through 11, and average those values to deduce the predicted rate during the 12th second. We continue this process of sliding the window rightward until we have predicted the request rate at time 270 seconds, based on the initial 10 seconds window.

If the estimated request rate at second 270 exceeds the current request rate, we determine the number of additional servers needed to meet the SLA (via the $k = \lceil R/r \rceil$ formula) and turn these on at time 11, so that they will be ready to run at time 270. If the estimated request rate at second 270 is lower than the current request rate, we look at the maximum request rate, M, during the interval from time 11 to time 270. If M is lower than the current request rate, then we turn off as many servers as we can while meeting the SLA for request rate M. Of course, the window size affects the performance of *MWA*. *We empower* MWA *by using the best window size for each trace.*

Figure 4.5(a) shows that the performance of *Predictive MWA* is very similar to what we saw for *Reactive*: low P_{avg} and N_{avg} values, beating *AlwaysOn* by a factor of 2, but high T_{95} values, typically exceeding the SLA by a factor of 10 to 20.



(a) T_{95}=7,740ms, P_{avg}=1,276W, N_{avg}=6.3 (b) T_{95}=2,544ms, P_{avg}=2,161W, N_{avg}=11.8

Figure 4.5: (a) *Predictive: MWA* and (b) *Predictive: LR*.

Predictive - Linear Regression (LR)

The *LR* policy is identical to *MWA* except that, to estimate the request rate at time 270 seconds, we use linear regression to match the best linear fit to the values in the window. Then we extend our line out by 260 seconds to get a prediction of the request rate at time 270 seconds. We now follow the same procedure as in *MWA* to determine whether we need to turn servers on or off based on these request rate estimates.

The performance of *Predictive LR* is worse than that of *Predictive MWA*. Response times are still bad, but now capacity and power consumption can be bad as well. The problem, as illustrated in Figure 4.5(b), is that the linear slope fit used in *LR* can end up overshooting the required capacity greatly.

4.3.5 AutoScale−−

One might think that the poor performance of the dynamic server provisioning policies we have seen so far stems from the fact that they are too slow to turn servers on when needed. However, an equally big concern is the fact that these policies are quick to turn servers *off* when not needed, and hence do not have those servers available when load subsequently rises. This rashness is particularly problematic in the case of bursty workloads, such as those in Figure 4.1.

AutoScale−− addresses the problem of *scaling down* capacity by being very conservative in turning servers off while doing nothing new with respect to turning servers

Trace \ t_{wait}		60s	120s	260s
Dual phase [144]	T_{95}	503ms	491ms	445ms
	P_{avg}	1,253W	1,297W	1,490W
	N_{avg}	7.0	7.2	8.8

Table 4.1: The (in)sensitivity of *AutoScale*--'s performance to t_{wait}.

on (the turning on algorithm is the same as in *Reactive*). We will show that *by simply taking more care in turning servers off*, *AutoScale*-- is able to outperform all the prior dynamic server provisioning policies we have seen with respect to meeting SLAs, while simultaneously keeping P_{avg} and N_{avg} low.

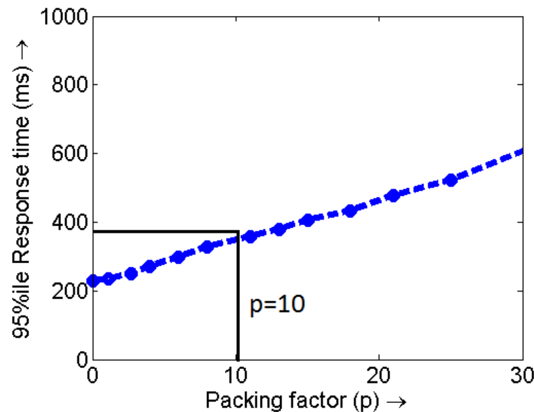
When to turn a server off?

Under *AutoScale*--, each server decides *autonomously* when to turn off. When a server goes idle, rather than turning off immediately, it sets a timer of duration t_{wait} and sits in the idle state for t_{wait} seconds. If a request arrives at the server during these t_{wait} seconds, then the server goes back to the busy state (with zero setup cost); otherwise the server is turned off. In our experiments for *AutoScale*--, we use a t_{wait} value of 120s. Table 4.1 shows that *AutoScale*-- is largely insensitive to t_{wait} in the range $t_{wait} = 60s$ to $t_{wait} = 260s$. There is a slight increase in P_{avg} (and N_{avg}) and a slight decrease in T_{95} when t_{wait} increases, due to idle servers staying on longer.

The idea of setting a timer before turning off an idle server has been proposed before (see, for example, [100, 125, 96]), however, *only for a single server*. For a multi-server system, *independently* setting timers for each server can be inefficient, since we can end up with too many idle servers. Thus, we need a more coordinated approach for using timers in our multi-server system which takes routing into account, as explained below.

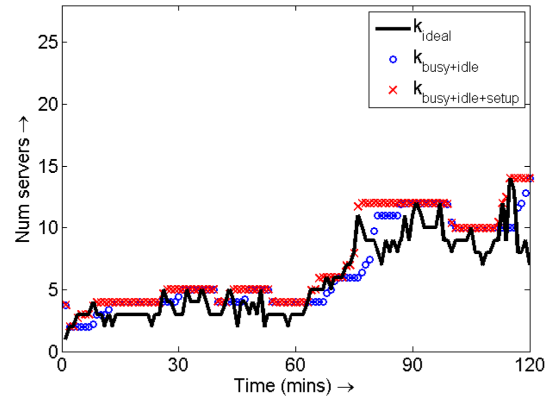
How to route requests to servers?

Timers prevent the mistake of turning off a server just before a new arrival comes in. However, they can also waste power and capacity by leaving too many servers in the idle state. We would basically like to keep only a small number of servers (just the *right* number) in the idle state.



95%ile response time vs. packing factor

Figure 4.6: For a single server, packing factor, $p = 10$.



T_{95}=491ms, P_{avg}=1,297W, N_{avg}=7.2

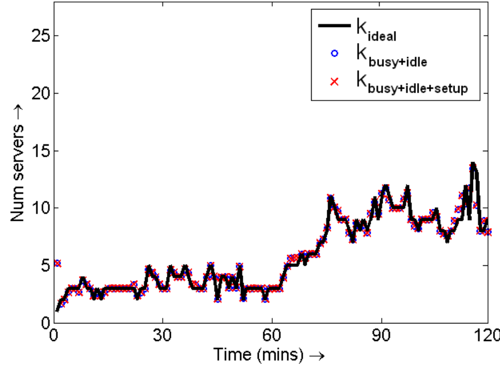
Figure 4.7: *AutoScale*−−.

To do this, we introduce a routing scheme that tends to concentrate requests onto a small number of servers, so that the remaining (unneeded) servers will naturally "time out." Our routing scheme uses an *index-packing* idea, whereby all on servers are indexed from 1 to n. Then we send each request to the lowest-numbered on server that currently has fewer than p requests, where p stands for *packing factor* and denotes the maximum number of requests that a server can serve concurrently and meet its response time SLA. For example, in Figure 4.6, we see that to meet a 95%ile guarantee of 400 ms, the packing factor is $p = 10$ (in general, the value of p depends on the system in consideration). When all on servers are already packed with p requests each, additional request arrivals are routed to servers via the join-the-shortest-queue routing.

The idea of combining timers with index-packing routing stems from theoretical work, where we prove [64] that, under more restrictive M/M/k models, this combination achieves the well-known "square-root staffing" [97] optimal capacity provisioning. *This is far more effective than adding some x% extra capacity, as in Section 4.3.3.*

In comparison with all the other policies, *AutoScale*−− hits the "sweet spot" of low T_{95} as well as low P_{avg} and N_{avg}. As seen from Table 4.2, *AutoScale*−− is close to the response time SLA in all traces except for the Big spike trace. Simultaneously, the mean power usage and capacity under *AutoScale*−− is typically significantly better than *AlwaysOn*, saving as much as a factor of two in power and capacity.

Figure 4.7 illustrates how *AutoScale*−− is able to achieve these performance results. Observe that the crosses and circles in *AutoScale*−− form flat constant lines,



T_{95}=320ms, P_{avg}=1,132W, N_{avg}=5.9

Figure 4.8: *Opt*.

instead of bouncing up and down, erratically, as in the earlier policies. This comes from a combination of the t_{wait} timer and the index-based routing, which together keep the number of servers just slightly above what is needed, while also avoiding toggling the servers between on and off states when the load goes up and down. Comparing Figures 4.7 and 4.4(b), we see that the combination of timers and index-based routing is far more effective than using *Reactive* with extra capacity, as in Section 4.3.3.

4.3.6 Opt

As a yardstick for measuring the effectiveness of *AutoScale*−−, we define an optimal policy, *Opt*, which behaves identically to *Reactive*, but with a setup time of zero. Thus, as soon as the request rate changes, *Opt* reacts by immediately adding or removing the required capacity, without having to wait for setup.

Figure 4.8 shows that under *Opt*, the number of servers on scales exactly with the incoming request load. *Opt* easily meets the T_{95} SLA, and consumes very little power and resources (servers). Note that while *Opt* usually has a T_{95} of about 320-350ms, and thus it might seem like *Opt* is over-provisioning, it just about meets the T_{95} SLA for the Steep tri phase trace (see Table 4.2) and hence cannot be made more aggressive.

In support of *AutoScale*−−, we find in Table 4.2 that *Opt*'s power consumption and server usage is only 30% less than that of *AutoScale*−−, averaged across all traces, despite *AutoScale*−− having to cope with the 260s setup time.

Trace \ Policy		*AlwaysOn*	*Reactive*	*Predictive MWA*	*Predictive LR*	*Opt*	*AutoScale--*
Slowly varying [95]	T_{95}	271ms	673ms	3,464ms	618ms	366ms	435ms
	P_{avg}	2,205W	842W	825W	964W	788W	1,393W
	N_{avg}	14.0	4.1	4.1	4.9	4.0	5.8
Quickly varying	T_{95}	303ms	20,005ms	3,335ms	12,553ms	325ms	362ms
	P_{avg}	2,476W	1,922W	2,065W	3,622W	1,531W	2,205W
	N_{avg}	14.0	10.1	10.6	22.1	8.2	15.1
Big spike [144]	T_{95}	229ms	3,426ms	9,337ms	1,753ms	352ms	854ms
	P_{avg}	2,260W	985W	998W	1,503W	845W	1,129W
	N_{avg}	14.0	4.9	4.9	8.1	4.5	6.6
Dual phase [144]	T_{95}	291ms	11,003ms	7,740ms	2,544ms	320ms	491ms
	P_{avg}	2,323W	1,281W	1,276W	2,161W	1,132W	1,297W
	N_{avg}	14.0	6.2	6.3	11.8	5.9	7.2
Large variations [144]	T_{95}	289ms	4,227ms	13,399ms	20,631ms	321ms	474ms
	P_{avg}	2,363W	1,391W	1,461W	2,576W	1,222W	1,642W
	N_{avg}	14.0	7.8	8.1	16.4	7.1	10.5
Steep tri phase [61]	T_{95}	377ms	$>$ 1 min	$>$ 1 min	661ms	446ms	463ms
	P_{avg}	2,263W	849W	1,287W	3,374W	1,004W	1,601W
	N_{avg}	14.0	5.2	7.2	20.5	5.1	8.0

Table 4.2: Comparison of all policies. Setup time = 260s throughout.

4.4 Wear-and-tear costs of server provisioning

Thus far we have evaluated server provisioning policies based on the metrics of T_{95}, P_{avg}, and N_{avg}. However, another metric that data center operators often care about is the wear-and-tear cost of servers. Dynamically provisioning servers increases their wear-and-tear [121, 120, 60, 129], and might also increase the risk of hardware failure [29]. Recent work [29, 121] suggests that a server should not be power cycled more than once per hour. We now evaluate our server provisioning policies based on the average frequency of power cycling a server per hour. We obtain the average frequency by dividing the total number of server power-on instances by the number of servers (28 in our case), and then dividing by the length of the trace (2 hours, except for the Quickly varying trace, which has a length of 0.5 hours).

The *Reactive* policy has an average power cycling frequency of 1.04 across all six traces, with a maximum of 3.21 for the Quickly varying trace and a minimum of 0.34

for the Slowly varying trace. The *MWA* and *LR* policies have an average frequency of 1.04 and 3.68 respectively, with a maximum of 3.14 and 9 respectively, for the Quickly varying trace, and a minimum of 0.23 and 0.43 respectively, for the Slowly varying trace. The Quickly varying trace results in higher power cycling frequency because of the numerous oscillations in request rate. The Slowly varying trace results in lower frequency because it exhibits only one significant oscillation in request rate.

The *AutoScale*-- policy has an average power cycling frequency of only 0.28, with a maximum of 0.36 for the Quickly varying trace and a minimum of 0.2 for the Slowly varying trace. This is to be expected because of the t_{wait} idea employed by *AutoScale*--, which significantly reduces unneeded power cycling. Based on the above results, we conclude that *AutoScale*-- is superior to the existing dynamic provisioning policies considered in this paper based on the power cycling frequency. Further, *AutoScale*-- necessitates, on average, only 0.28 power cycles for a server per hour, which is much lower than the suggested limit of one power cycle per hour. Note that the static provisioning policy, *AlwaysOn*, is superior, by definition, to any dynamic provisioning policy since it does not power cycle servers.

The above discussion considers the average power cycling frequency across all servers. However, it is also important to take into account the distribution of power cycling across the servers. In practice, the distribution can be made more uniform across all servers by taking account *which* server is being turned on. For example, when an additional server is to be provisioned, we can pick the off server that has the lowest power cycling frequency. This decision will not affect the values of T_{95}, P_{avg}, N_{avg}, or the mean power cycling frequency.

4.5 Impact of Sleep States

Sleep states are essentially low-power inactive states, and can be considered as "efficient" off states. A sleep state can be defined by its setup time and the power consumed when in sleep. The setup time for exiting a sleep state is typically less that the setup time for exiting the off state. However, the power consumed when the server is sleeping is non-zero, though the sleep power is typically much less than the idle power consumption. While sleep states have existed for mobile devices and desktop computers for some time, they have largely not been incorporated into the servers in today's data centers. High setup times make data center administrators fearful of any form of dynamic server provisioning, whereby servers are suspended or shut down when load drops. This general reluctance has stalled research into whether

there might be *some* feasible sleep state (with sufficiently low setup overhead and/or sufficiently low power) that would actually be beneficial in data centers.

Prior work [86, 107, 168, 51, 192, 125] evaluated dynamic power management using only *existing* sleep states in desktop computers, mobile class computers (such as laptops and notebooks), custom built servers, and sub-systems such as processors and microdrives. Given the limited range of existing sleep states, and because sleep states on different systems may look very different, it is difficult to assess the full potential of sleep states. While there has been some work [131, 132] considering the effectiveness of hypothetical sleep states, the hypothetical states considered in [131, 132] are limited to transition times (setup times) on the order of 1 millisecond, and thus do not span the space of what is realistically possible today in servers.

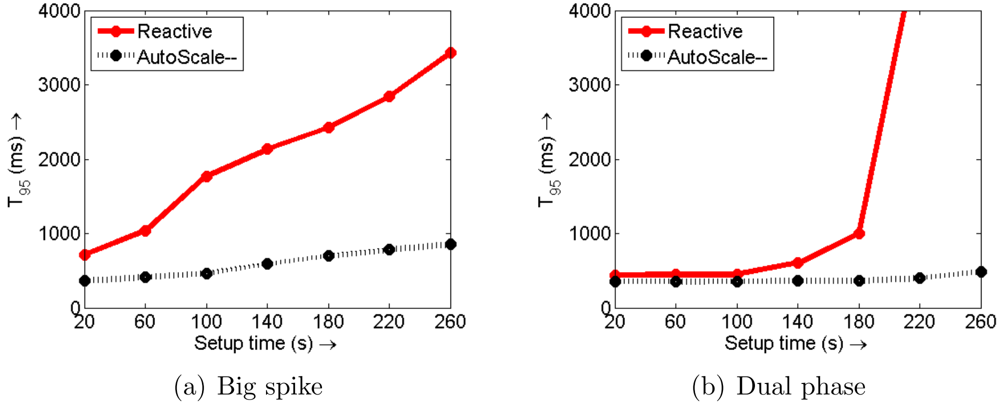
In this section we contrast the performance of *AutoScale*-- with other dynamic server provisioning policies, in the presence of sleep states. We look at lower setup times in Section 4.5.1 and lower sleep power in Section 4.5.2. We then provide a sensitivity analysis of sleep states in Section 4.5.3. This analysis helps identify sleep states that are useful for dynamic server provisioning. In our experiments, we achieve lower setup times and lower sleep power by tweaking our implementation testbed as discussed in Section 2.3.2. Also, for *AutoScale*--, we reduce the value of t_{wait} in proportion to the reduction in setup time.

The results presented here are also applicable to virtual machines, which typically require a smaller setup time for provisioning than physical servers. These results will also be useful for future server architectures which may support sleep states.

4.5.1 Lower setup times

When the setup time is very low, approaching zero, then by definition, all policies approach *Opt*. For moderate setup times, one might expect that *AutoScale*-- does not provide significant benefits over other policies such as *Reactive*, since T_{95} should not rise too much during the setup time. This turns out to be false since the T_{95} under *Reactive* continues to be high even for moderate setup times.

Figure 4.9(a) shows our experimental results for T_{95} for the Big spike trace [144], under *Reactive* and *AutoScale*--. We see that as the setup time drops, the T_{95} drops almost linearly for both *Reactive* and *AutoScale*--. However, *AutoScale*-- continues to be superior to *Reactive* with respect to T_{95} for any given setup time. In fact, even when the setup time is only 20s, the T_{95} under *Reactive* is almost twice that under *AutoScale*--. This is because of the huge spike in load in the Big spike trace



(a) Big spike (b) Dual phase

Figure 4.9: Effect of lower setup times for (a) Big spike trace [144] and (b) Dual phase trace [144].

that cannot be handled by *Reactive* even at low setup times. We find similar results for the Steep tri phase trace [61], with T_{95} under *Reactive* being more than three times as high as that under *AutoScale*--. The P_{avg} and N_{avg} values for *Reactive* and *AutoScale*-- also drop with setup time, but these changes are not as significant as the changes for T_{95}.

Figure 4.9(b) shows our experimental results for T_{95} for the Dual phase trace [144], under *Reactive* and *AutoScale*--. This time, we see that as the setup time drops below 100s, the T_{95} under *Reactive* approaches that under *AutoScale*--. This is because of the relatively small fluctuations in load in the Dual phase trace, which can be handled by *Reactive* once the setup time is small enough. However, for setup times larger than 100s, *AutoScale*-- continues to be significantly better than *Reactive*. We find similar results for the Quickly varying trace and the Large variations trace [144]. Note that the T_{95} under *AutoScale*-- is only slightly affected by the setup time because *AutoScale*-- fundamentally works by *avoiding setup times*.

Thus, depending on the trace, *Reactive* can perform poorly even for low setup times (see Figure 4.9(a)). We expect similar behavior under the *Predictive* policies as well. Note that *AlwaysOn* and *Opt* are not affected by setup times. We summarize the results of this section by making the following observation:

Observation 4.1 (Dynamic server provisioning under lower setup times)**.** *Our dynamic server provisioning policy,* AutoScale--, *is beneficial not only for high setup times but also for more moderate setup times. When the setup time is very low, all dynamic server provisioning policies approach optimality.*

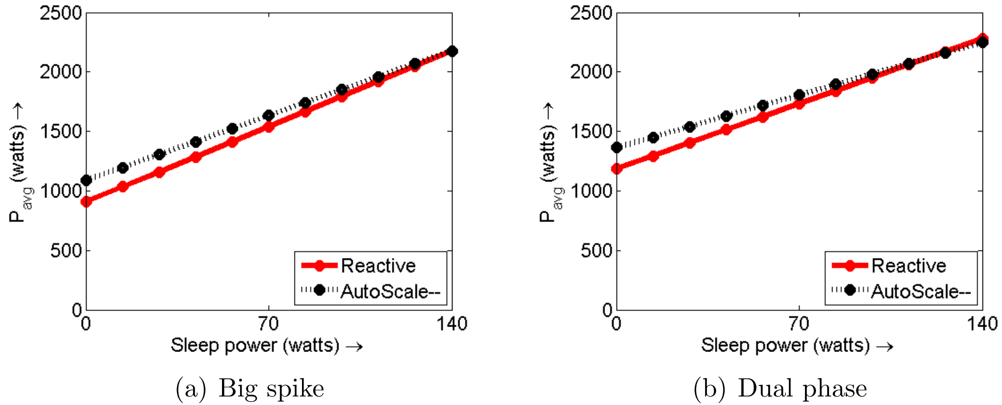


Figure 4.10: Effect of lower sleep power for (a) Big spike trace [144] and (b) Dual phase trace [144]. Here, the setup time is 200s.

4.5.2 Lower sleep power

Figures 4.10(a) and 4.10(b) show our experimental results for P_{avg} for the Big spike trace [144] and the Dual phase trace [144] respectively, under *Reactive* and *AutoScale*−−. These results are for a setup time of 200s. We see that the P_{avg} value for *Reactive* and *AutoScale*−− drops almost linearly with the sleep power. This is to be expected since the percentage of time that the servers are sleeping is constant for a given setup time and a given trace. Thus, a drop in sleep power lowers the power consumption of the servers proportionately. We see that *Reactive* has a lower P_{avg} than *AutoScale*−− for lower sleep powers. This is consistent with our results in Table 4.2 using the off state which has zero "sleep" power. However, over the entire range of the sleep power, the difference in P_{avg} for *Reactive* and *AutoScale*−− is fairly small. We find similar results for lower setup times as well. Note that the T_{95} value is not affected by the sleep power and is thus not shown. We summarize our results with the following observation:

Observation 4.2 (Dynamic server provisioning under lower sleep power)**.**
The system power consumption under dynamic server provisioning policies drops almost linearly with sleep power.

4.5.3 Sensitivity analysis of sleep states

In this section we provide a sensitivity analysis of sleep states. Recall that a sleep state can be defined by its setup time and the power consumed when in sleep. We examine sleep states with setup times ranging from 20s to 200s, and sleep power ranging from 0W to 140W (idle power). Our goal is to identify regimes of sleep states which are useful for dynamic server provisioning policies, specifically, *Reactive* and *AutoScale*--. We use Performance-per-Watt, or PPW, to evaluate sleep states. Recall from Definition 2.6 that:

$$\text{PPW} = \frac{1}{T_{95} \cdot P_{avg}}$$

Using this metric, we define a sleep state to be "useful" for a dynamic server provisioning policy if the PPW for that policy when using the sleep state is greater than the PPW of *AlwaysOn*. Note that *AlwaysOn* does not make use of sleep states. To compare the policies, we look at the Normalized Performance-per-Watt, NPPW, defined as the PPW for the dynamic policy, say *Reactive*, normalized by the PPW for *AlwaysOn*:

$$\text{NPPW} = \frac{\text{PPW}^{Reactive}}{\text{PPW}^{AlwaysOn}}$$

When NPPW exceeds 1, we say that *Reactive* is superior to *AlwaysOn*.

Figures 4.11 and 4.12 show our experimental results for NPPW for the Big spike trace [144] and the Dual phase trace [144], respectively, under (a) *Reactive* and (b) *AutoScale*--. For additional experimental results on other traces, we refer the reader to our papers [71, 72]. In these figures, the light shaded regions indicate regimes of sleep states where the dynamic policy is superior to *AlwaysOn* (NPPW > 1), and the dark shaded regions indicate regimes of sleep states where the dynamic policy is inferior to *AlwaysOn* (NPPW < 1). The solid red line indicates the cross-over points where NPPW=1. In general, NPPW increases as the setup time decreases or as the sleep power decreases.

From the results, we see that the usefulness of sleep states significantly depends on the trace. For example, sleep states with a setup time $\leq$ 120s and sleep power $\leq$ 42W are useful for *Reactive* under the Dual phase trace in Figure 4.12(a). However, for the Big spike trace in Figure 4.11(a), even our best sleep state with a setup time of 20s and sleep power of 0W results in an NPPW of only 0.7 for *Reactive*. Interestingly, the effect of setup time and sleep power on NPPW depends on the variability of

Setup time (s) ↓ / Sleep power (watts) →	0	14	28	42	56	70	84	98	112	126	140
200	0.2	0.2	0.2	0.2	0.2	0.1	0.1	0.1	0.1	0.1	0.1
180	0.2	0.2	0.2	0.2	0.2	0.1	0.1	0.1	0.1	0.1	0.1
160	0.2	0.2	0.2	0.2	0.1	0.1	0.1	0.1	0.1	0.1	0.1
140	0.3	0.2	0.2	0.2	0.2	0.2	0.1	0.1	0.1	0.1	0.1
120	0.3	0.3	0.2	0.2	0.2	0.2	0.2	0.1	0.1	0.1	0.1
100	0.3	0.3	0.3	0.2	0.2	0.2	0.2	0.2	0.2	0.1	0.1
80	0.4	0.3	0.3	0.3	0.2	0.2	0.2	0.2	0.2	0.2	0.2
60	0.6	0.5	0.5	0.4	0.4	0.3	0.3	0.3	0.3	0.2	0.2
40	0.7	0.6	0.6	0.5	0.5	0.4	0.4	0.4	0.3	0.3	0.3
20	0.7	0.6	0.6	0.5	0.5	0.4	0.4	0.4	0.3	0.3	0.3

(a) *Reactive*

Setup time (s) ↓ / Sleep power (watts) →	0	14	28	42	56	70	84	98	112	126	140
200	0.5	0.5	0.4	0.4	0.4	0.4	0.3	0.3	0.3	0.3	0.3
180	0.6	0.5	0.5	0.5	0.4	0.4	0.4	0.4	0.3	0.3	0.3
160	0.7	0.6	0.6	0.5	0.5	0.5	0.4	0.4	0.4	0.4	0.3
140	0.7	0.7	0.6	0.6	0.5	0.5	0.5	0.5	0.4	0.4	0.4
120	1.0	0.9	0.8	0.8	0.7	0.7	0.6	0.6	0.5	0.5	0.5
100	1.1	1.0	0.9	0.8	0.8	0.7	0.7	0.6	0.6	0.5	0.5
80	1.2	1.0	0.9	0.9	0.8	0.7	0.7	0.6	0.6	0.6	0.5
60	1.2	1.1	1.0	0.9	0.8	0.8	0.7	0.7	0.7	0.6	0.6
40	1.3	1.3	1.1	1.0	1.0	0.9	0.8	0.8	0.7	0.7	0.6
20	1.3	1.2	1.1	1.0	0.9	0.9	0.8	0.8	0.7	0.7	0.7

(b) *AutoScale--*

Figure 4.11: NPPW for the Big spike trace [144] under (a) *Reactive* and (b) *AutoScale--*. The solid red line indicates the cross-over points where NPPW=1.

Setup time (s) ↓ / Sleep power (watts) →	0	14	28	42	56	70	84	98	112	126	140
200	0.3	0.3	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.1
180	0.6	0.5	0.5	0.5	0.4	0.4	0.4	0.4	0.3	0.3	0.3
160	0.7	0.7	0.6	0.6	0.5	0.5	0.5	0.4	0.4	0.4	0.4
140	1.0	0.9	0.8	0.8	0.7	0.7	0.6	0.6	0.6	0.5	0.5
120	1.3	1.2	1.1	1.0	0.9	0.8	0.8	0.7	0.7	0.7	0.6
100	1.3	1.2	1.1	1.0	0.9	0.9	0.8	0.8	0.7	0.7	0.7
80	1.4	1.3	1.2	1.1	1.0	0.9	0.9	0.8	0.8	0.7	0.7
60	1.4	1.2	1.1	1.0	1.0	0.9	0.8	0.8	0.7	0.7	0.7
40	1.4	1.3	1.2	1.1	1.0	0.9	0.9	0.8	0.8	0.7	0.7
20	1.4	1.3	1.2	1.1	1.0	0.9	0.9	0.8	0.8	0.7	0.7

(a) *Reactive*

Setup time (s) ↓ / Sleep power (watts) →	0	14	28	42	56	70	84	98	112	126	140
200	1.3	1.3	1.2	1.1	1.1	1.0	1.0	0.9	0.9	0.9	0.8
180	1.4	1.3	1.2	1.2	1.1	1.0	1.0	0.9	0.9	0.9	0.8
160	1.4	1.3	1.2	1.2	1.1	1.1	1.0	1.0	0.9	0.9	0.8
140	1.4	1.3	1.2	1.2	1.1	1.1	1.0	1.0	0.9	0.9	0.8
120	1.4	1.3	1.3	1.2	1.1	1.1	1.0	1.0	0.9	0.9	0.8
100	1.5	1.4	1.3	1.2	1.1	1.1	1.0	1.0	0.9	0.9	0.8
80	1.5	1.4	1.3	1.2	1.2	1.1	1.0	1.0	0.9	0.9	0.9
60	1.6	1.5	1.4	1.3	1.2	1.1	1.1	1.0	0.9	0.9	0.9
40	1.6	1.5	1.4	1.3	1.2	1.1	1.1	1.0	0.9	0.9	0.9
20	1.6	1.5	1.4	1.3	1.2	1.1	1.1	1.0	1.0	0.9	0.9

(b) *AutoScale--*

Figure 4.12: NPPW for the Dual phase trace [144] under (a) *Reactive* and (b) *AutoScale--*. The solid red line indicates the cross-over points where NPPW=1.

request rate in the trace. For example, for *AutoScale--* under the Big spike trace in Figure 4.11(b), the NPPW depends on both the setup time and the sleep power. However, under the Dual phase trace in Figure 4.12(b), the NPPW for *AutoScale--* is largely insensitive to the setup time.

We also see that *AutoScale--* is superior to *Reactive* in terms of NPPW for all sleep states that we consider. *AutoScale--* is also superior to *AlwaysOn* for most of the sleep states under the Dual phase trace. However, for the Big spike trace, *AutoScale--* is superior to *AlwaysOn* for only a few sleep states. This result highlights the fact that *dynamic server provisioning is hindered by load spikes, even when using sleep states.*

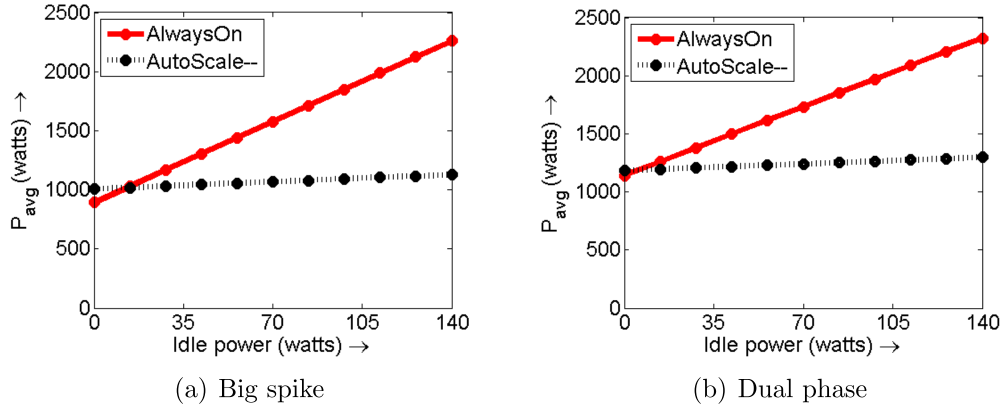
Effect of scale on sleep states

Thus far, we have only looked at the usefulness of sleep states on our implementation testbed. In order to investigate the effect of scale (size of the testbed) on the usefulness of sleep states, we resort to simulations and analytical models. While not shown here, our results indicate that the usefulness of sleep states (NPPW) increases with scale. This is because as the size of the data center goes up, the probability that all servers are simultaneously busy goes down. Thus, an incoming request has higher chances of finding an idle server, thereby lowering T_{95} and increasing PPW for *Reactive* and *AutoScale*--. By contrast, for the over-provisioned *AlwaysOn*, T_{95} is always good whereas P_{avg} is always high, regardless of the data center size. The net effect is an increase in NPPW for *Reactive* and *AutoScale*--. Of course, the improvement in NPPW cannot go on forever. There is a natural lower bound on T_{95}, namely the T_{95} provided by *AlwaysOn*. Once we reach this lower bound on T_{95}, NPPW cannot improve further. Interestingly, the NPPW for both *Reactive* and *AutoScale*-- converges to roughly the same value as we scale the size of the data center, meaning that *Reactive* approaches *AutoScale*-- for large scale data centers. We now end this section with a summary of the above findings:

Observation 4.3 (Usefulness of sleep states)**.**
Dynamic server provisioning policies equipped with "useful" sleep states are often superior to static policies like AlwaysOn*. The regime of useful sleep states for the dynamic policies, however, depends on the variability in the demand traces. Interestingly, the usefulness of sleep states improves with the size of the data center.*

4.6 Impact of Lower Idle Power

With advances in processor technology, it is very likely that the server idle power will drop. This claim is also supported by recent literature [69, 131, 132]. The drop in server idle power should greatly benefit the static server provisioning policy, *AlwaysOn*, by lowering its P_{avg}, since a lot of servers are idle under *AlwaysOn*. However, for the dynamic server provisioning policies, we expect P_{avg} to only drop slightly, since servers are rarely idle under such policies. To explore the effects of lower server idle power, we contrast the performance of *AutoScale*-- with that of *AlwaysOn*. The idle power for the servers in our testbed is about 140W (with C-states enabled), as mentioned in Section 2.3.2. We replicate the effects of lower idle power by tweaking our testbed along the same lines as discussed in Section 2.3.2.



(a) Big spike

(b) Dual phase

Figure 4.13: Effect of lower idle power for (a) Big spike trace [144] and (b) Dual phase trace [144]. Here, the setup time is 260s.

Figures 4.13(a) and 4.13(b) show our experimental results for P_{avg} for the Big spike trace [144] and the Dual phase trace [144] respectively, under *AlwaysOn* and *AutoScale*−−. Here, the setup time is 260s. We see that the P_{avg} value for *AlwaysOn* drops almost linearly with the server idle power. This is to be expected since the number of servers idle under *AlwaysOn* for a given trace is constant, and thus, a drop in server idle power lowers the power consumption of these idle servers proportionately. The P_{avg} value for *AutoScale*−− also drops with the idle power, but this drop is negligible. It is interesting to note that the P_{avg} value for *AlwaysOn* drops below that of *AutoScale*−− only when the idle power is extremely low (less than 15W). We find similar results for the other traces as well. Note that we are being particularly conservative in assuming that while the server idle power drops, the power consumed by the servers when they are in setup remains the same. This assumption hurts the P_{avg} value for *AutoScale*−−. The T_{95} value is not affected by the server idle power and is thus not shown. We conclude this section with the following observation:

Observation 4.4 (Dynamic server provisioning under lower idle power)**.**
While lower idle power consumption favors static provisioning policies like AlwaysOn*, the power savings under our dynamic server provisioning policy,* AutoScale−−*, continue to be greater than those under* AlwaysOn*, except when the idle power is extremely low.*

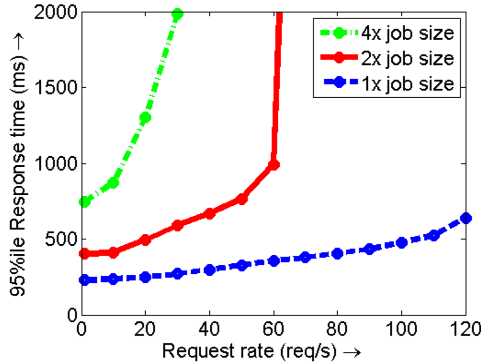
4.7 Evaluation II: Robustness

Thus far in our traces we have only varied the request rate over time. However, in reality there are many other ways in which load can change. For example, if new features or security checks are added to the application, the request size might increase. We mimic such effects by increasing the number of key-value lookups associated with each request. As a second example, if any abnormalities occur in the system, such as internal service disruptions, slow networks, or maintenance cycles, servers may respond more slowly, and requests may accumulate at the servers. We mimic such effects by slowing down the frequency of the application servers. All the dynamic server provisioning policies described thus far, with the exception of *Opt*, use the request rate to scale capacity. However, using the request rate to determine the required capacity is somewhat *fragile*. If the request *size* increases, or if servers become *slower*, due to any of the reasons mentioned above, then the number of servers needed to maintain acceptable response times ought to be increased. In both cases, however, no additional capacity will be provisioned if the policies only look at request rate to scale up capacity.

4.7.1 Why request rate is not a good feedback signal

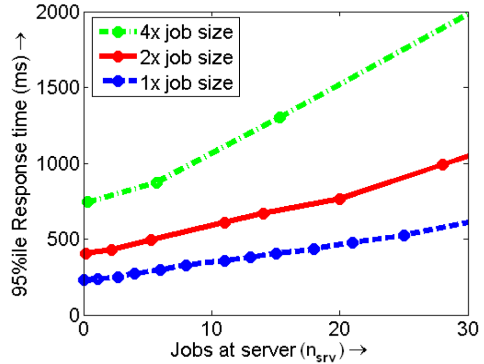
In order to assess the limitations of using request rate as a feedback signal for scaling capacity, we ran *AutoScale*-- on the Dual phase trace with a 2x request size (meaning that our request size is now 240ms as opposed to the 120ms size we have used thus far). Since *AutoScale*-- does not detect an increase in request size, and thus does not provision for this, its T_{95} shoots up ($T_{95} = 51,601ms$). This is also true for the *Reactive* and *Predictive* policies, as can be seen in Tables 4.4 and 4.5 for the case of increased request size and in Table 4.6 for the case of slower servers.

Figure 4.14 shows measured 95%ile response time at a single server versus request rate for different request sizes. It is clear that while each server can handle 60 req/s without violating the T_{95} SLA for a 1x request size, the T_{95} shoots up for the 2x and 4x request sizes. An obvious way to solve this problem is to determine the request size. However, it is not easy to determine the request size since the size is usually not known ahead of time. Trying to derive the request size by monitoring the response times does not help either, since response times are usually affected by queueing delays. Thus, we need to come up with a better feedback signal than request rate or request size.



95%ile response time vs. request rate

Figure 4.14: A single server can no longer handle 60 req/s when the request size increases.



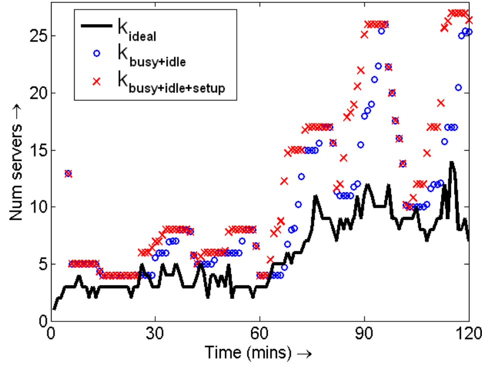
95%ile response time vs. n_{srv}

Figure 4.15: For a single server, setting $n_{srv} = p = 10$ works well for all request sizes.

4.7.2 A better feedback signal that is still not quite right

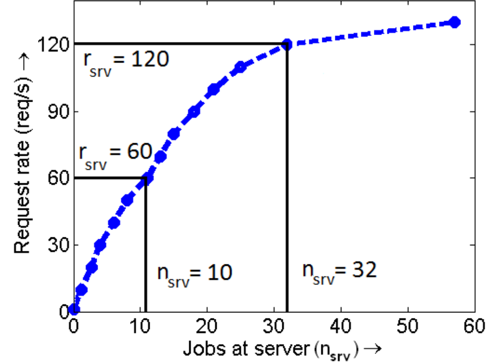
We propose using the number of requests in the system, n_{sys}, as the feedback signal for scaling up capacity rather than the request rate. We assert that n_{sys} more faithfully captures the dynamic state of the system than the request rate: If the system is under-provisioned *either* because the request rate is too high *or* because the request size is too big *or* because the servers have slowed down, n_{sys} will tend to increase. If the system is over-provisioned, n_{sys} will tend to decrease below some expected level. Further, calculating n_{sys} is fairly straightforward; many modern systems (including our Apache load balancer) already track this value, and it is instantaneously available.

Figure 4.15 shows the measured 95%ile response time at a single server versus the number of requests at a single server, n_{srv}, for different request sizes. Note that $n_{srv} = n_{sys}$ in the case of a single-server system. Surprisingly, the 95%ile response time values do not shoot up for the 2x and 4x request sizes for a given n_{srv} value. In fact, setting $n_{srv} = 10$, as in Section 4.3.5, provides acceptable T_{95} values for all request sizes (note that T_{95} values for the 2x and 4x request sizes are higher than 500ms, which is to be expected as the work associated with each request is naturally higher). This is because an increase in the request size (or a decrease in the server speed) increases the rate at which "work" comes into each server. This increase in work is reflected in the consequent increase in n_{srv}. By limiting n_{srv} using p, the packing factor (the maximum number of requests that a server can serve concurrently and meet its SLA), we can limit the rate at which work comes in to



T_{95}=441ms, P_{avg}=2,083W, N_{avg}=12.5

Figure 4.16: Our proposed policy overshoots while scaling up capacity, resulting in high P_{avg} and N_{avg}.



Request rate vs. number of requests

Figure 4.17: A doubling of request rate can lead to a tripling of number of requests at a single server.

each server, thereby adjusting the required capacity to ensure that we meet the T_{95} SLA. Based on these observations, we set $p = 10$ for the 2x and 4x request sizes. Thus, *p is agnostic to request sizes for our system*, and only needs to be computed once. The insensitivity of p to request sizes is to be expected since p represents the degree of parallelism for a server, and thus depends on the specifications of a server (number of cores, hyper-threading, etc), and not on the request size.

Based on our observations from Figure 4.15, we propose a plausible solution for dynamic server provisioning based on looking at the total number of requests in the system, n_{sys}, as opposed to looking at the request rate. The idea is to provision capacity to ensure that the number of requests at a server is $n_{srv} = 10$. In particular, the proposed policy is exactly the same as *AutoScale*−−, except that it estimates the required capacity as $k_{reqd} = \lceil n_{sys}/10 \rceil$, where n_{sys} is the total number of requests in the system at that time. In our implementation, we sample n_{sys} every 20 seconds, and thus, the proposed policy re-scales capacity, if needed, every 20 seconds. Note that the proposed policy uses the same method to scale down capacity as *AutoScale*−−, viz., using a timeout of 120s along with the index-packing routing.

Figure 4.16 shows how our proposed policy behaves for the 1x request size. We see that our proposed policy successfully meets the T_{95} SLA, but it clearly overshoots in terms of scaling up capacity when the request rate goes up. Thus, the proposed policy results in high power and resource consumption. One might think that this overshoot can be avoided by packing more requests at each server, thus allowing n_{srv} to be higher than 10. However, note that the T_{95} in Figure 4.16 is already quite close

to the 500ms SLA, and increasing the number of requests packed at a server beyond 10 can result in SLA violations.

Figure 4.17 explains the overshoot in terms of scaling up capacity for our proposed policy. We see that when the request rate into a single server, r_{srv}, doubles from 60 req/s to 120 req/s, n_{srv} more than doubles from 10 to 32. This is because an overloaded server ($r_{srv} = 120$ req/s) slows down due to an increase in context switches and due to other side-effects such as trashing, leading to an increase in the number of outstanding requests (n_{srv}). Thus, our proposed policy scales up capacity by a factor of 3, whereas ideally capacity should only be scaled up by a factor of 2. Clearly our proposed policy does not work so well, even when the request size is just 1x.

We now introduce our *AutoScale* policy, which solves our problems of scaling up capacity.

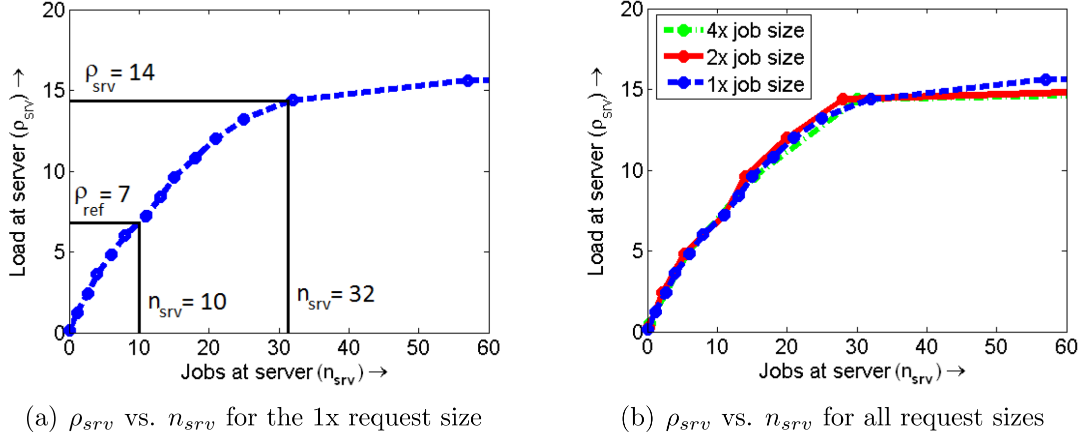
4.7.3 AutoScale: Incorporating the right feedback signal

We now describe the *AutoScale* policy and show that it not only handles the case where request rate changes, but also handles cases where the request size changes (see Tables 4.4 and 4.5) or where the server efficiency changes (see Table 4.6).

AutoScale differs from the server provisioning policies described thus far in that it uses the number of requests in the system, n_{sys}, as the feedback signal rather than request rate. However, *AutoScale* does not simply scale up the capacity linearly with an increase in n_{sys}, as was the case with our proposed policy above. This is because n_{sys} grows *super-linearly* during the time that the system is under-provisioned, as is well known in queueing theory [160, Chapter 8]. Instead, *AutoScale* tries to infer the *amount of work* in the system by monitoring n_{sys}. The amount of work in the system is proportional to both the request rate and the request size (the request size in turn depends also on the server efficiency), and thus, we try to infer the product of request rate and request size, which we call *system load*, ρ_{sys}. Formally,

$$\rho_{sys} = \begin{array}{c}\text{request rate into}\\ \text{the data center } (R)\end{array} \times \begin{array}{c}\text{average}\\ \text{request size,}\end{array}$$

where the average 1x request size is 120ms. Fortunately, there is an easy relationship (which we describe soon) between the number of requests in the system, n_{sys}, and the system load, ρ_{sys}, obviating the need to ever measure load or request rate or the request size. Once we have ρ_{sys}, it is easy to get to the required capacity, k_{reqd}, since ρ_{sys} represents the amount of work in the system and is hence proportional to k_{reqd}.



(a) ρ_{srv} vs. n_{srv} for the 1x request size (b) ρ_{srv} vs. n_{srv} for all request sizes

Figure 4.18: Load at a server as a function of the number of requests at a server for various request sizes. Surprisingly, the graph is invariant to changes in request size.

We now explain the translation process from n_{sys} to ρ_{sys}, and then from ρ_{sys} to k_{reqd}. We refer to this entire translation algorithm as the *capacity inference algorithm*. The full translation from n_{sys} to k_{reqd} will be given in Equation (4.4) below. A full listing of all the variables used in this section is provided in Table 7.1 for convenience.

The capacity inference algorithm

In order to understand the relationship between n_{sys} and ρ_{sys}, we first derive the relationship between the number of requests at a single server, n_{srv}, and the load at a single server, ρ_{srv}. Formally, the load at a server is defined as

$$\rho_{srv} = \begin{array}{c}\text{request rate into}\\ \text{a single server } (r_{srv})\end{array} \times \begin{array}{c}\text{average}\\ \text{request size,}\end{array} \tag{4.1}$$

where the average 1x request size is 120ms and r_{srv} is the request rate into a single server. If the request rate to a server, r_{srv}, is made as high as possible without violating the SLA, then the resulting ρ_{srv} from Equation (4.1) is referred to as the reference load, ρ_{ref}. For our system, recall that the maximum request rate into a single server without violating the SLA is $r_{srv} = 60$ req/s (see Figure 4.2). Thus,

$$\rho_{ref} = 60 \times 0.12 \approx 7, \tag{4.2}$$

meaning that a single server can handle a load of at most 7 without violating the SLA, assuming a 1x request size of 120ms.

Variable	Description
r_{srv}	Request rate into a single server
R	Request rate into the data center
n_{sys}	Number of requests in the system
n_{srv}	Number of requests at a server
p	Packing factor (maximum n_{srv} without violating SLA)
ρ_{sys}	System load
ρ_{srv}	Load at a server
ρ_{ref}	Reference load (for a single server)
k_{reqd}	Required capacity (number of servers)
k_{curr}	Current capacity

Table 4.3: Description of variables.

Returning to the discussion of how ρ_{srv} and n_{srv} are related, we expect that ρ_{srv} should increase with n_{srv}. Figure 4.18(a) shows our experimental results for ρ_{srv} as a function of n_{srv}. Note that $\rho_{srv} = \rho_{ref} = 7$ corresponds to $n_{srv} = p = 10$, where p is the packing factor. We obtain Figure 4.18(a) by converting r_{srv} in Figure 4.17 to ρ_{srv} using Equation (4.1) above. Observe that when ρ_{srv} doubles from 7 to 14, we see that n_{srv} more than triples from 10 to 32, as was the case in Figure 4.17.

We will now estimate ρ_{sys}, the system load, using the relationship between n_{srv} and ρ_{srv}. To estimate ρ_{sys}, we first approximate n_{srv} as $\frac{n_{sys}}{k_{curr}}$, where k_{curr} is the current number of on servers. We then use n_{srv} in Figure 4.18(a) to estimate the corresponding ρ_{srv}. Finally, we have $\rho_{sys} = k_{curr} \times \rho_{srv}$. In summary, given the number of requests in the system, n_{sys}, we can derive the system load, ρ_{sys}, as follows:

$$\boldsymbol{n_{sys}} \xrightarrow{\div k_{curr}} n_{srv} \xrightarrow{Fig.\ 4.18(a)} \rho_{srv} \xrightarrow{\times k_{curr}} \boldsymbol{\rho_{sys}} \tag{4.3}$$

Surprisingly, the relationship between the number of requests at a server, n_{srv}, and the load at a server, ρ_{srv}, does not change when request size changes. Figure 4.18(b) shows our experimental results for the relationship between n_{srv} and ρ_{srv} for different request sizes. We see that the plot is invariant to changes in request size. Thus, while calculating $\rho_{sys} = k_{curr} \times \rho_{srv}$, we do not have to worry about the request size and we can simply use Figure 4.18(a) to estimate ρ_{sys} from n_{sys} *irrespective of the request size*. Likewise, we find that the relationship between n_{srv} and ρ_{srv} does not change when the server speed changes. This is because a decrease in server speed is the same as an increase in request size for our system.

The reason why the relationship between n_{srv} and ρ_{srv} is agnostic to request size is because ρ_{srv}, by definition (see Equation (4.1)), takes the request size into account. If the request size doubles, then the request rate into a server needs to drop by a factor of 2 in order to maintain the same ρ_{srv}. These changes result in exactly the same *amount of work entering the system per unit time*, and thus, n_{srv} does not change. The insensitivity of the relationship between n_{srv} and ρ_{srv} to changes in request size is consistent with queueing-theoretic analysis [103]. Interestingly, this insensitivity, coupled with the fact that the packing factor, p, is a constant for our system ($p = 10$, see Section 4.7.2), results in *the reference load, ρ_{ref}, being a constant for our system*, since $\rho_{ref} = \rho_{srv}$ for the case when $n_{srv} = p = 10$ (see Figure 4.18(a)). Thus, we only need to compute ρ_{ref} once for our system.

Now that we have ρ_{sys} from Equation (4.3), we can translate this to the required capacity, k_{reqd}, using ρ_{ref}. Since ρ_{sys} corresponds to the total system load, while ρ_{ref} corresponds to the load that a single server can handle, we deduce that the required capacity is:

$$k_{reqd} = \left\lceil \frac{\rho_{sys}}{\rho_{ref}} \right\rceil$$

In summary, we can get from n_{sys} to k_{reqd} by first translating n_{sys} to ρ_{sys}, which leads us to k_{reqd}, as outlined below:

$$\boldsymbol{n_{sys}} \xrightarrow{\div k_{curr}} n_{srv} \xrightarrow{Fig.\ 4.18(a)} \rho_{srv} \xrightarrow{\times k_{curr}} \rho_{sys} \xrightarrow{\div \rho_{ref}} \boldsymbol{k_{reqd}} \quad (4.4)$$

For example, if $n_{sys} = 320$ and $k_{curr} = 10$, then we get $n_{srv} = 32$, and from Figure 4.18(a), $\rho_{srv} = 14$, irrespective of request size. The load for the system, ρ_{sys}, is then given by $k_{curr} \times \rho_{srv} = 140$, and since $\rho_{ref} = 7$, the required capacity is $k_{reqd} = \lceil k_{curr} \times \frac{\rho_{srv}}{\rho_{ref}} \rceil = 20$. Consequently, *AutoScale* turns on 10 additional servers. In our implementation, we reevaluate k_{reqd} every 20s to avoid excessive changes in the number of servers.

The insensitivity to request size of the relationship between n_{srv} and ρ_{srv} from Figure 4.18(b) allows us to use Equation (4.4) to compute the desired capacity, k_{reqd}, in response to any form of load change. Further, as noted above, p and ρ_{ref} are constants for our system, and only need to be computed once.

The design of *AutoScale* includes a few key parameters: t_{wait} (see Table 4.1), p (derived in Figure 4.6), ρ_{ref} (derived in Equation (4.2)), and the ρ_{srv} vs. n_{srv} relationship (derived in Figure 4.18(a)). In order to deploy *AutoScale* on a given cluster, these parameters need to be determined. Fortunately, all of the above parameters only need to be determined once for a given cluster. This is because these parameters

Trace \ Policy		*AlwaysOn*	*Reactive*	*Predictive MWA*	*Predictive LR*	*Opt*	*AutoScale*
Slowly varying [95]	T_{95}	478ms	> 1 min	> 1 min	> 1 min	531ms	701ms
	P_{avg}	2,127W	541W	597W	728W	667W	923W
	N_{avg}	14.0	3.2	2.7	3.8	4.0	5.4
Dual phase [144]	T_{95}	424ms	> 1 min	> 1 min	> 1 min	532ms	726ms
	P_{avg}	2,190W	603W	678W	1,306W	996W	1,324W
	N_{avg}	14.0	3.0	2.6	6.6	5.8	7.3

Table 4.4: Comparison of all policies for 2x request size[1].

Trace \ Policy		*AlwaysOn*	*Reactive*	*Predictive MWA*	*Predictive LR*	*Opt*	*AutoScale*
Slowly varying [95]	T_{95}	759ms	> 1 min	> 1 min	> 1 min	915ms	1,155ms
	P_{avg}	2,095W	280W	315W	391W	630W	977W
	N_{avg}	14.0	1.9	1.7	2.1	4.0	5.7
Dual phase [144]	T_{95}	733ms	> 1 min	> 1 min	> 1 min	920ms	1,217ms
	P_{avg}	2,165W	340W	389W	656W	985W	1,304W
	N_{avg}	14.0	1.7	1.8	3.2	5.9	7.2

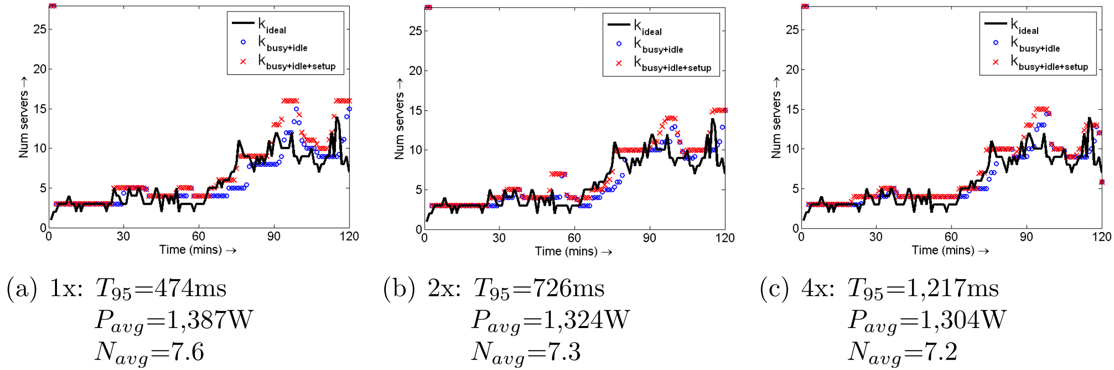
Table 4.5: Comparison of all policies for 4x request size[1].

depend on the specifications of the system, such as the server type, the setup time, and the application, which do not change at runtime. Request rate, request size, and server speed, can all change at runtime, but these do not affect the value of the above key parameters. This makes *AutoScale* a very robust provisioning policy.

Performance of AutoScale

Tables 4.4 and 4.5 summarize our experimental results for the case where the number of key-value lookups per request (or the request size) increases by a factor of 2 and 4 respectively. Because request sizes are dramatically larger, and because the number

[1]For a given arrival trace, when request size is scaled up, the size of the application tier should ideally be scaled up as well so as to accommodate the increased load. However, since our application tier is limited to 28 servers, we follow up an increase in request size with a proportionate decrease in request rate for the arrival trace. Thus, the peak load (request rate times request size) is the same before and after the request size increase, and our 28 server application tier suffices for the experiment. In particular, *AlwaysOn*, which knows the peak load ahead of time, is able to handle peak load by keeping 14 servers on even as the request size increases.



(a) 1x: T_{95}=474ms P_{avg}=1,387W N_{avg}=7.6

(b) 2x: T_{95}=726ms P_{avg}=1,324W N_{avg}=7.3

(c) 4x: T_{95}=1,217ms P_{avg}=1,304W N_{avg}=7.2

Figure 4.19: Robustness of *AutoScale* to changes in request size. The request size is 1x (or 120ms) in (a), 2x (or 240ms) in (b), and 4x (or 480ms) in (c).

of servers in our testbed is limited, we compensate for the increase in request size by scaling down the request rate by the same factor. Thus, in Table 4.4, request sizes are a factor of two larger than in Table 4.2, but the request rate is half that of Table 4.2. The T_{95} values are expected to increase as compared with Table 4.2 because each request now takes longer to complete (since it does more key-value lookups).

Looking at *AutoScale* in Table 4.4, we see that T_{95} increases to around 700ms, while in Table 4.5, it increases to around 1200ms. This is to be expected. By contrast, for all other dynamic server provisioning policies, the T_{95} values exceed one minute, both in Tables 4.4 and 4.5. Again, this is because these policies react only to changes in the request rate, and thus end up typically under-provisioning. We do not show the results for *AutoScale*-- in Tables 4.4 and 4.5, but its performance is just as bad as the other dynamic server provisioning policies that react only to changes in the request rate. *AlwaysOn* knows the peak load ahead of time, and thus, always keeps N_{avg} = 14 servers on. As expected, the T_{95} values for *AlwaysOn* are quite good, but P_{avg} and N_{avg} are very high. Comparing *AutoScale* and *Opt*, we see that *Opt*'s power consumption and server usage is again only about 30% less than that of *AutoScale*.

Figure 4.19 shows the server behavior under *AutoScale* for the Dual phase trace for request sizes of 1x, 2x and 4x. Clearly, *AutoScale* is successful at handling the changes in load due to both changes in request rate and changes in request size.

Table 4.6 illustrates another way in which load can change. Here, we return to the 1x request size, but this time all servers have been slowed down to a frequency of 1.6 GHz as compared with the default frequency of 2.26 GHz. By slowing down the frequency of the servers, T_{95} naturally increases. We find that for all the dynamic

Trace \ Policy		*AlwaysOn*	*Reactive*	*Predictive MWA*	*Predictive LR*	*Opt*	*AutoScale*
Slowly varying [95]	T_{95}	572ms	> 1 min	> 1 min	3,339ms	524ms	760ms
	P_{avg}	2,132W	903W	945W	863W	638W	1,123W
	N_{avg}	14.0	5.7	5.9	4.8	4.0	7.2
Dual phase [144]	T_{95}	362ms	24,401ms	23,412ms	2,527ms	485ms	564ms
	P_{avg}	2,147W	1,210W	1,240W	2,058W	1,027W	1,756W
	N_{avg}	14.0	6.3	7.4	12.2	5.9	10.8

Table 4.6: Comparison of all policies for lower CPU frequency.

server provisioning policies, except for *AutoScale*, the T_{95} shoots up. The reason is that these other dynamic server provisioning policies provision capacity based on the request rate. Since the request rate has not changed as compared to Table 4.2, they typically end up under-provisioning, now that servers are slower. The T_{95} for *AlwaysOn* does not shoot up because even in Table 4.2, it is greatly over-provisioning by provisioning for the peak load at all times. Since the *AutoScale* policy is robust to all changes in load, it provisions correctly, resulting in acceptable T_{95} values. P_{avg} and N_{avg} values for *AutoScale* continue to be much lower than that of *AlwaysOn*, similar to Table 4.2.

Tables 4.4, 4.5 and 4.6 clearly indicate the superior robustness of *AutoScale* which uses n_{sys} to respond to changes in load, allowing *AutoScale* to respond to all forms of changes in load.

4.7.4 Alternative feedback signal choices

AutoScale employs the number of requests in the system, n_{sys}, as opposed to request rate, as the feedback signal for provisioning capacity. An alternative feedback signal that we could have employed in *AutoScale* is T_{95}, the performance metric. Using the performance metric as a feedback signal is a popular choice in control-theoretic approaches [117, 124, 114, 141]. While using T_{95} as the feedback signal might allow *AutoScale* to achieve the same robustness properties as provided by n_{sys}, we would first have to come up with an analogous capacity inference algorithm for the T_{95} feedback signal. As discussed in Section 4.7.2, using an inaccurate capacity inference algorithm can result in poor server provisioning. For single-tier systems, one can use simple empirical models or analytical approximations to derive the capacity inference algorithm, as was the case in [117, 124, 41]. However, for multi-tier systems, coming up with a capacity inference algorithm can be quite difficult, as noted in [141].

Further, performance metrics such as T_{95} depend not only on the system load, ρ_{sys}, but also on the request size, as is well known in queueing theory [103]. Thus, the capacity inference algorithm for the T_{95} feedback signal will not be invariant to request size.

Other choices for the feedback signal that have been used in prior work include system-level metrics such as CPU utilization [74, 86, 117], memory utilization [74], network bandwidth [117], etc. A major drawback of employing these feedback signals, as mentioned in [86], is that utilization and bandwidth values saturate at 100%, and thus, the degree of under-provisioning cannot be determined via these signals alone. This makes it difficult to derive a capacity inference algorithm for these feedback signals.

4.8 Prior Work

We discussed the prior work in the broad area of dynamic server provisioning in Section 2.2. In this section we discuss prior work that is directly related to the scope of *AutoScale*, and the specific prior work that *AutoScale* builds upon.

Krioukov et al. [107] use various predictive policies, such as Last Arrival, *MWA*, Exponentially Weighted Average, and *LR*, to predict the future request rate (to account for setup time), and then accordingly add or remove servers from a heterogenous pool. The authors find that *MWA* and *LR* work best for the traces they consider (Wikipedia.org traffic), providing significant power savings over *AlwaysOn*. Motivated by this observation, we chose *MWA* and *LR* as the representative predictive policies for comparison with *AutoScale*. The *AlwaysOn* version used by Krioukov et al. does not know the peak request rate ahead of time (in fact, in many experiments they set *AlwaysOn* to provision for twice the historically observed peak), and is thus not as powerful an adversary as the version we employ.

Chen et al. [40] use auto-regression techniques to predict the request rate for a seasonal arrival pattern, and then accordingly turn servers on and off using a simple threshold policy. While the setup in [40] is very different (seasonal arrival patterns) from our own, there is one similarity to *AutoScale* in their approach: like *AutoScale*, the authors in [40] use the index-based routing (see Section 4.3.5). However, the policy in [40] does not have any of the robustness properties of *AutoScale*, nor the t_{wait} timeout idea.

Hoffman et al. [85] consider a single-tier system with unpredictable load fluctuations. The authors employ a reactive approach using the quality of service (for

example, the bitrate or image quality) as the feedback signal to create a robust system. However, the workload considered by the authors allows for a loss in quality of service, thus obviating the need to scale capacity during load fluctuations. In our system, we do not have any leeway on the quality of service since we have a strict T_{95} SLA. Thus, during load fluctuations, *AutoScale* must dynamically scale capacity to maintain the required SLA.

While there have been a lot of approaches to dynamic server provisioning (see Section 2.2), most of these approaches focus only on changes in request rate. We are not aware of any dynamic server provisioning approaches that have considered changes in request size or server efficiency for multi-tier applications.

4.9 Chapter Summary

This chapter considers dynamic server provisioning policies in the presence of unpredictable load and setup costs. We find that existing reactive approaches that simply scale capacity based on the current request rate are too rash to turn servers off, especially when request rate is bursty. Given the huge setup time needed to turn servers back on (typically on the order of minutes [93, 129]), response times suffer greatly when request rate suddenly rises. Predictive approaches that work well when request rate is periodic or seasonal perform very poorly in our case where traffic is unpredictable. Furthermore, as we show in Section 4.3.3, leaving a fixed buffer of extra capacity is also not the right solution.

Our solution, *AutoScale*, takes a fundamentally different approach to dynamic server provisioning. First, *AutoScale* does not try to predict the future request rate. Instead, *AutoScale* introduces a smart policy to automatically provision spare servers, which can absorb unpredictable changes in request rate. We make the case that to successfully meet response time SLAs, it suffices to simply manage existing capacity carefully and not give away spare capacity recklessly (see Table 4.2). Existing reactive approaches can be easily modified to be more conservative in giving away spare capacity so as to inherit *AutoScale*'s ability to absorb unpredictable changes in request rate. Second, *AutoScale* is able to handle unpredictable changes not just in the request rate but also unpredictable changes in the request size (see Tables 4.4 and 4.5) and the server efficiency (see Table 4.6). *AutoScale* does this by provisioning capacity using not the request rate, but rather the number of requests in the system, which it is able to translate into the correct capacity via a novel, non-trivial algorithm. As illustrated via our experimental results in Tables 4.2 to 4.6, *AutoScale*

outclasses existing optimized predictive and reactive policies in terms of consistently meeting response time SLAs. While *AutoScale*'s 95%ile response time numbers are usually less than one second, the 95%ile response times of existing predictive and reactive policies often exceed one full minute!

Not only does *AutoScale* allow us to save power while meeting response time SLAs, but it also allows us to save on rental costs when leasing virtual resources from cloud service providers by reducing the amount of resources needed to successfully meet response time SLAs.

While one might think that *AutoScale* will become less valuable as setup times decrease (due to, for example, sleep states or virtual machines), or as server idle power decreases (due to, for example, advances in processor technology), we find that this is not the case. *AutoScale* can significantly lower response times and power consumption when compared to existing policies even for low setup times (see Figure 4.9) and reasonably low idle power (see Figure 4.13). In fact, even when the setup time is only 20s, *AutoScale* can lower 95%ile response times by a factor of 3.

AutoScale was designed to overcome the challenges presented by unpredictable demand. Most of the unpredictability in workload demand can be attributed to short-term fluctuations in demand (see Chapter 3), which are successfully handled by *AutoScale*. However, unpredictability in demand can also be caused by load spikes, which hinder *AutoScale*. We see the detrimental effects of load spikes in Table 4.2 under the Big spike trace. In the next chapter, we provide a solution, *SoftScale*, for handling load spikes. *SoftScale* is easily integrated into *AutoScale*, thus making *AutoScale* robust to load spikes. The combination of *AutoScale* and *SoftScale* represents a dynamic provisioning policy that successfully overcomes unpredictability in demand.

Chapter 5

SoftScale: A Novel Approach to Handling Load Spikes

In this chapter we address the challenges presented by load spikes. Abrupt changes in load, or load spikes, are very problematic since they result in a very steep rise in response time (see Observation 3.7). While dynamic policies like *AutoScale* are good at handling fluctuations in demand, they are still plagued by load spikes. In fact, even if load spikes are instantaneously detected and the required additional capacity is provisioned within 5 seconds, response time can still be severely affected. This is illustrated by our experimental results in Figures 3.13 and 3.14.

We present *SoftScale*, a practical approach to handling load spikes in multi-tier data centers without having to over-provision resources. *SoftScale* works by opportunistically stealing resources from other tiers to alleviate the bottleneck tier, even when the tiers are carefully provisioned at capacity. *SoftScale* is especially useful during the transient overload periods when additional capacity is being brought online. Importantly, *SoftScale* can be used in conjunction with existing dynamic server provisioning policies, such as *AutoScale*.

We introduce the problem and discuss the scope of this chapter in Section 5.1. We then describe our experimental setup for this chapter in Section 5.2. We present the design and implementation of *SoftScale* in Section 5.3, and evaluate *SoftScale* under a range of load spikes, including artificial instantaneous load spikes, load spikes in real traces, and load spikes created by system failures, in Section 5.4. We investigate the applicability of *SoftScale* under lower setup times in Section 5.5, and under future many-core processors in Section 5.6. We discuss prior work in Section 5.7 and conclude with a summary of this chapter in Section 5.8.

5.1 Introduction

Data centers play an important role in today's IT infrastructure. Government organizations, hospitals, financial trading firms, and major IT companies, such as Google, Facebook and Amazon, all rely on data centers for their daily business activities. A primary goal for data center operators is to provide good response times to users; these response time targets typically translate to some response time Service Level Agreements (SLAs). A secondary goal is to reduce operational costs by exploiting the variability in user demand. By scaling capacity to match current demand, operators can either:
(i) reduce power consumption by turning off unneeded servers, or
(ii) save on rental costs by releasing unneeded virtual machines, or
(iii) get additional work done by repurposing unneeded servers for other tasks.

Data center services today are often organized as multiple tiers. Typically, one of these tiers is an *application tier* that processes requests, and another tier is the *data tier* that is responsible for efficiently delivering data back to the application tier. While it is possible to physically colocate the application tier and the data tier on the same servers, dividing the architecture into physically different tiers is preferable because it makes it easier to scale and manage the individual tiers [52, 163, 173]. *The data tier is stateful, and is almost never turned off [172, 34], even if there is a significant drop in load [24].* The application tier, on the other hand, is usually stateless and can be dynamically scaled using existing reactive [114, 141, 148], predictive [107, 86] or mixed [173, 76, 60] approaches, *provided that the load does not change too abruptly.*

Unfortunately, abrupt changes in load, or load spikes, are all too common in today's data centers. Important events, such as the September 11 attacks [113, 88], earthquakes or other natural disasters [186], slashdot effects [7], Black Friday shopping [44], or sporting events, such as the Super Bowl [145] or the Soccer World Cup [20], are common causes of load spikes for website traffic. Service outages [147] or server failures [162] can also result in abrupt changes in load caused by a sharp drop in capacity. While some of the above events are predictable, most of them *cannot* be predicted in advance.

Abrupt changes in load are especially problematic since adding capacity requires some time, which we call *setup time* (see Definition 1.3), denoted by t_{setup}. Even if we instantaneously detect a spike in load, it will still take the system at least the setup time to add the required capacity. In our lab, the setup time for turning on an additional server is approximately 4-5 minutes. Similar setup times have also

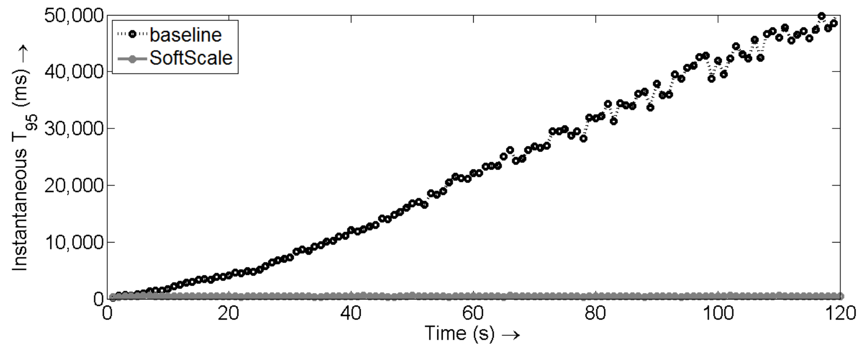


Figure 5.1: Using *SoftScale*, we can meet response time SLAs even under a 15% to 30% load jump. Note that the y-axis ranges from 0s to 50s.

been reported in recent literature [93, 129]. Likewise, the setup time for virtual machines (VMs) can range anywhere from 30 seconds – 1 minute if the VMs are locally created (based on our measurements using kvm [102]) or 2 – 10 minutes if the VMs are obtained from a cloud platform (see, for example, [13, 108]). All these numbers are extremely high, and can result in long periods where the SLA is violated.

Throughout this chapter, we focus on the performance of the system during the setup time following a load spike. Since no additional capacity can be added during the setup time, the system has a fixed number of servers online, and we refer to such a system as the *baseline*. A typical SLA requires that the 95th percentile of response time, denoted by T_{95}, stay below 500ms. In this chapter we consider the more difficult goal of meeting the T_{95} requirements during the setup time (i.e., after the onset of the spike, and before additional servers can be brought online). This is equivalent to saying that *no more than 5% of all requests that arrive during the setup time are allowed to exceed the 500ms response time*. In addition to the T_{95} (which measures over the entire setup time), in some plots, we also show the "instantaneous T_{95}" (see Definition 2.3), which is the 95th percentile of response times collected every second.

Consider a system which has the appropriate number of application servers turned on to ensure that the 95th percentile of response times stays below 500ms at the current load of 15% of peak load. Here, peak load refers to the maximum load that our system can handle (see Section 5.2 for details of our experimental testbed). Now, imagine that the load suddenly increases to 30%. The time needed to turn on the necessary additional servers is the setup time, say 5 minutes. We say that our system can "handle" a load jump if $T_{95} \leq 500ms$ during the setup time. As shown in Figure 5.1, our baseline system is not able to handle the 15% to 30% load jump. The black dots in Figure 5.1 show the increase in instantaneous T_{95}

during the first two minutes of the setup time under the baseline, where the system is clearly under-provisioned during this time. The data for Figure 5.1 is generated from experiments running on our implementation testbed using a key-value based workload (see Section 5.2 for full details of our experimental testbed). As shown in the figure, instantaneous T_{95} increases rapidly over time, reaching 50 seconds after only two minutes. Even if future hardware reduces this setup time to 10 seconds, we see that instantaneous T_{95} can be well over 3 seconds.

In order to avoid setup times, data center operators typically over-provision capacity at all times (since load spikes are often unpredictable). For example, to handle a 15% to 30% load jump, one needs to over-provision resources by a factor of 2. Clearly, such an approach is quite expensive.

We propose *SoftScale*, an approach that allows data centers to handle load spikes without having to over-provision resources and incur costs. *SoftScale* leverages the fact that the data tier in a multi-tier data center is always left on [172, 34, 24]. Thus, during the setup time following a load spike, we can use these "always on" data tier servers to do some of our application work. *SoftScale* involves running the application tier software on the data tier servers, where this software is only used during the setup time. We refer to this notion as "stealing" of the data tier capacity. *SoftScale* requires no additional resources and can even handle a doubling of load, so long as the final load is not too high. Returning to our example where the load instantaneously doubles from 15% to 30%, we see that *SoftScale*, denoted by the flat gray line in Figure 5.1, allows the instantaneous T_{95} to stay within the 500ms SLA at all times. While stealing from the data tier can increase the latency of data operations, the overall benefit of being able to meet SLAs during setup times makes a compelling case for using *SoftScale*. Note that one could theoretically use *SoftScale* even after the setup time, however, the (non-zero) increase in latency of data operations as a result of using *SoftScale* suggests otherwise. The *SoftScale* architecture is depicted in Figure 5.2, and is described in detail in Section 5.3.4.

Almost all papers on dynamic server provisioning (see, for example, [114, 141, 148, 107, 86, 173, 76, 60]) deal with new approaches to scale capacity in response to changes in load. However, such approaches can be ineffective during the setup time, as shown in Figure 5.1. *SoftScale* is a complementary solution that aims to improve performance *specifically during the setup time*, and is meant to be used in conjunction with any existing dynamic server provisioning approach.

While the concept behind *SoftScale* seems obvious, there are some practical difficulties that may have led researchers to dismiss this idea as "unworkable," hence the lack of publications on this idea. First, there is the question of *when* is *SoftScale*

useful. Since the data tier is provisioned to handle peak load, invoking *SoftScale* when the data tier is already bottlenecked will lead to SLA violations. Second, there is the question of *how much* can we steal from the data tier. If we end up stealing too much from the data tier, overall system performance might degrade. Third, there is the fear that running application work on the data tier servers will interfere with data delivery work, and can possibly lead to SLA violations. Finally, there is the fear that implementing *SoftScale* is too complicated.

In this chapter we demonstrate via implementation that *SoftScale* is a practical solution that allows us to meet response time SLAs even when load increases suddenly by a factor of 2, provided that the load is not too high. In particular, we make the following contributions:

- We determine *load regimes* for which *SoftScale* can be successfully applied to handle load spikes (see Section 5.3.1). This addresses the question of *when* to invoke *SoftScale*. Further, identifying load regimes where *SoftScale* is *not* beneficial avoids accidental overload of the data tier.
- We determine *how much* data tier capacity can be leveraged by *SoftScale* for a given load (see Section 5.3.2). This enables us to steal the *right amount* of capacity from the data tier without hurting overall response time.
- We show that it is possible to avoid interference between the application work and the data delivery work on the data servers by simply *isolating* these processes to different CPU cores (see Section 5.3.3).
- We outline the steps needed to implement the *SoftScale* middleware (see Section 5.3.4). In our testbed, we implemented *SoftScale* by adding less than a thousand lines of code in the Apache load balancer.
- We present an analytical model that estimates the system performance under *SoftScale* (see Section 5.3.5), allowing us to predict the performance of *SoftScale* for a range of multi-tier systems.

We evaluate *SoftScale* via implementation on a 28-server multi-tier testbed hosting a key-value based application built along the lines of Facebook or Amazon. Our implementation results show that *SoftScale* can be used to handle instantaneous load spikes (see Section 5.4.1), load spikes seen in real-world traces (see Section 5.4.2), as well as load spikes caused by server failures (see Section 5.4.3). To fully investigate the applicability of *SoftScale*, we experiment with multiple setup times ranging from 5 minutes (see Section 5.4) all the way down to 5 seconds (see Section 5.5). Our results indicate that *SoftScale* can provide huge benefits across the entire spectrum of setup times. We also investigate the applicability of *SoftScale* in future server

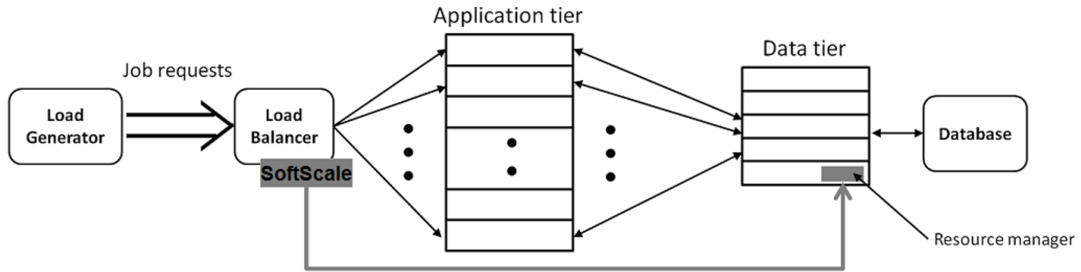


Figure 5.2: Our experimental testbed.

architectures which may have a larger number of CPU cores per server. Our results (see Section 5.6) indicate that *SoftScale* will be even more beneficial in such cases.

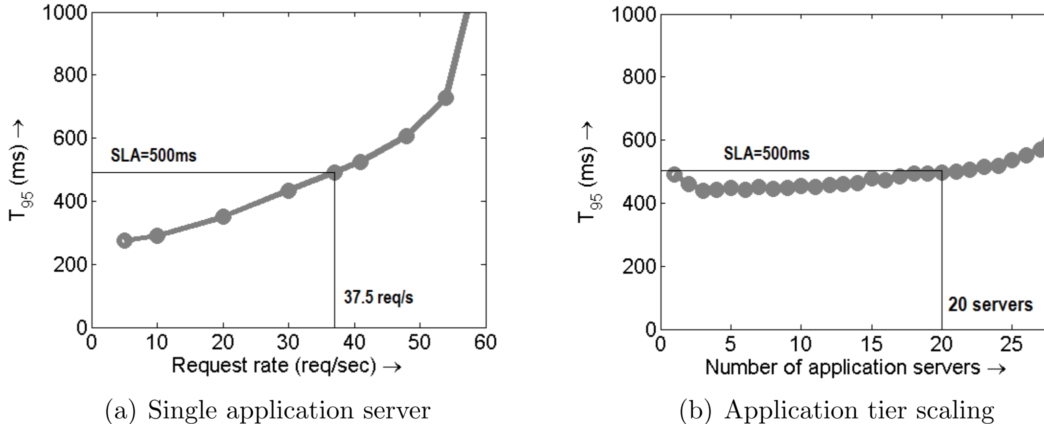
5.2 Our Experimental Testbed

Figure 5.2 illustrates our experimental testbed, which was described in Section 2.3.2. The gray components make up *SoftScale*, and will be described in detail in Section 5.3.4. While the data storage in our testbed includes the caching tier and the database, we only focus on leveraging the caching tier in this chapter. We refer to the caching tier as the "data tier" throughout this chapter.

We measure power consumption and use that as a proxy for all operational (resource) costs. We monitor the power consumption of individual servers by reading the power values from the power distribution unit. The idle power for our servers is about 140W (with C-states enabled) and the average power for our servers when they are busy or in setup is about 200W. The setup time for our servers is approximately $t_{setup} = 5$ minutes. However, we also examine the effects of lower t_{setup}.

5.2.1 Workload

For the experiments in this chapter, we modify our workload described in Section 2.3.2. Specifically, each workload request corresponds to an average of roughly 2,200 key-value fetches, which translates to a mean service time of approximately 200 ms, assuming no resource contention. We use the Zipf [143] distribution to model the popularity of requests. To minimize the effects of misses in the memcached layer (which could result in an unpredictable fraction of the requests violating the response time SLA), we tune the parameters of the Zipf distribution so that only a negligible fraction of requests miss in the memcached layer.



(a) Single application server (b) Application tier scaling

Figure 5.3: Figure (a) shows that a single application server can handle 37.5 req/s per server. Figure (b) shows that once we have more than 20 application servers, they can no longer handle 37.5 req/s per server because the memcached tier becomes the bottleneck.

5.2.2 Provisioning

In order to demonstrate the effectiveness of *SoftScale*, we tune our implementation testbed to have no spare capacity at the memcached tier at peak load. Our memcached tier comprises 5 servers, each with a 6-core Intel Xeon X5650 processor and 48GB of memory. However, we offline two cores[1] per server to be consistent with the specifications that were published by Facebook [128], leaving us with 4-core memcached servers. We now determine how many application servers we need to fully saturate the memcached tier.

Each of our application servers is a powerful 8-core (dual-socket) Intel Xeon E5520 processor-based server. We run an experiment where we have one application server and all five memcached servers, and we flood the system. We find that the application server can handle at most 37.5 req/s without violating the SLA, as shown in Figure 5.3(a).

We now examine how well the system scales as we add more application servers. Ideally, if we have x application servers, the system should be able to handle a maximum request rate of at least $37.5 \times x$ req/s without violating the 500ms SLA. Figure 5.3(b) shows our scaling results, where we vary the number application servers

[1]Observe that weakening the memcached servers greatly hurts *SoftScale* in that there is less capacity to steal, but we do this purposely to create a fully saturated memcached tier.

from 1 to 28, and use a request rate of 37.5 req/s times the number of application servers. We see that the system scales perfectly up to 20 application servers. Once we have more than 20 application servers, we see that they can no longer handle 37.5 req/s per server. This is because at this *peak load*, which corresponds to $37.5 \times 20 = 750$ req/s, the memcached tier starts becoming a bottleneck. We validate our claim by ensuring that the other components in the system, namely the load generator, the load balancer, and the application servers, are not a bottleneck. Further, by monitoring the network bandwidth, we ensure that it is not a bottleneck. With this ratio of 20 application servers to 5 memcached servers, we ensure that the memcached tier is saturated. Thus, at least 5 memcached servers are needed to handle peak load (using more than 5 memcached servers only improves the performance of *SoftScale*). This 4:1 ratio of application servers to memcached servers is consistent with Facebook [128].

Based on the above experiments, we conclude that the 5 memcached servers can handle at most 750 req/s. Thus, in our experiments, we limit our total request rate to 750 req/s, which we also refer to as peak load or 100% load. At peak load, there is no spare capacity in the memcached tier. Thus, we cannot "steal" any resources from memcached servers at high load without violating the 500ms SLA.

When running the system, the 5 memcached servers are always kept on. By contrast, the number of application servers needed at any time is $\lceil \frac{r}{37.5} \rceil$, where r is the current request rate into the system. For example, if the current request rate is 15% of the peak (or 112 req/s), we provision $\lceil \frac{112}{37.5} \rceil = 3$ application servers. Now, if the load suddenly doubles from 15% (112 req/s) to 30% (225 req/s), we need 6 application servers in total. Thus, the 3 application servers that are currently on become the bottleneck (until the additional application servers complete setting up).

Note that the ratio of application servers to memcached servers is no longer 4:1 if the application tier is scaled down. For example, if the current request rate is 25% of the peak (or 187 req/s), then we provision $\lceil \frac{187}{37.5} \rceil = 5$ application servers. This gives us a 1:1 ratio of application servers to memcached servers.

5.3 SoftScale

The key idea behind *SoftScale* is to leverage the computational power at the always on data tier servers to do some of our application work during the setup time while additional application tier capacity is being brought online. The motivation behind this idea is that, while our memcached servers are provisioned to have exactly the

right amount of resources at high load (for our system, peak load is 750 req/s), there are extra resources available at low load. Thus, when the system load is low, we should be able to "steal" resources from the always on memcached servers to offset some of the workload at the bottlenecked application servers.

SoftScale works by enhancing the Apache load balancer to route some of the application requests to the memcached servers during load spikes. Note that the software needed to process the application work will first have to be installed on the data tier servers. For our experimental testbed, this only involved installing the Apache web server with PHP support on the memcached servers. Further, our application software does not consume a lot of memory.

While *SoftScale* sounds like a promising idea, exploiting the full potential of *SoftScale* is challenging. We now describe *SoftScale* by discussing the design decisions behind the algorithm.

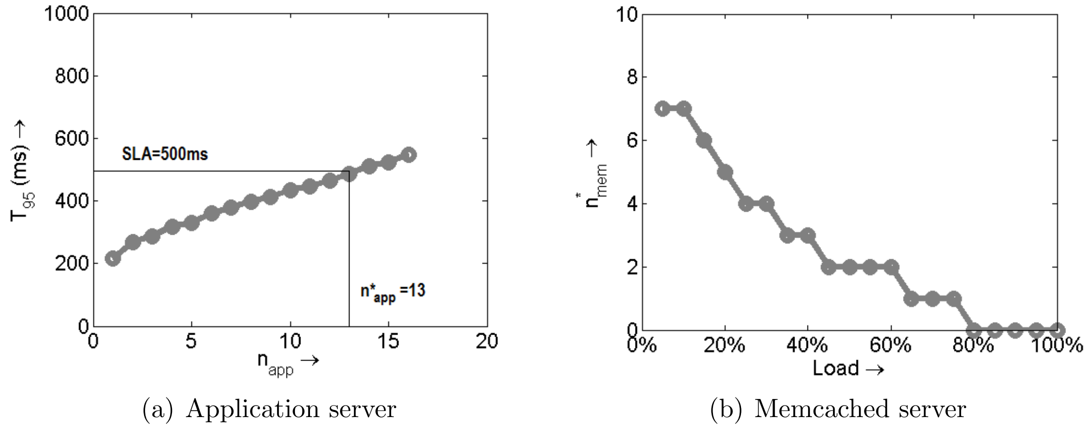
5.3.1 When to invoke SoftScale?

SoftScale must be invoked *as soon as there is a spike in load*. A spike in load could be caused *either* by an increase in request rate *or* by a loss in application tier capacity (server failures or service outages).

If the spike in load is caused by a sudden increase in request rate, then the obvious approach to detect this spike would be to monitor request rate periodically. Unfortunately, request rate is a time-average value, and is thus not instantaneous enough to detect load spikes. We propose monitoring the *number of active requests* at each application server, n_{app}, to detect load spikes. If the system is under-provisioned because the request rate is too high, then n_{app} will immediately increase. Monitoring n_{app} is fairly straightforward, and many modern systems, including the Apache load balancer, already track this value.

Spikes in load can also be caused by a sudden loss in application tier capacity (server failures or service outages). In this case, request rate cannot be used to detect the spike. Fortunately, n_{app} is immediately responsive to server failures, since it increases instantaneously when the application tier capacity drops.

We must invoke *SoftScale* when n_{app} becomes so high that the T_{95} SLA is in danger of being violated. In particular, if n^*_{app} is the maximum number of simultaneous requests that a single application server can handle without violating the SLA, then we invoke *SoftScale* as soon as n_{app} exceeds n^*_{app} for all application servers. Of course, one can also be conservative and invoke *SoftScale* even when n_{app} is below n^*_{app}.



(a) Application server (b) Memcached server

Figure 5.4: Figure (a) shows that we should invoke *SoftScale* whenever the number of requests at the application server exceeds 13. Figure (b) shows n^*_{mem}, the optimal number of application requests that can be simultaneously handled by a memcached server without violating the 500ms SLA, as a function of the total system load.

An easy way to determine n^*_{app} is by profiling the application servers. We run a closed-loop experiment with a single application server where we fix the number of simultaneous requests in the system (n_{app}), and monitor T_{95}. Figure 5.4(a) shows our results. We see that, for our system, $n^*_{app} = 13$. This same technique (profiling the application servers) can be used for determining n^*_{app} for different systems as well. Note that n^*_{app} corresponds to the 37.5 req/s that each application server can handle. Since we provision the application tier so as not to exceed 37.5 req/s at each server, *a reading of* $n_{app} > 13$ *indicates overload*. Thus, we invoke *SoftScale* as soon as the load balancer detects that n_{app} has exceeded 13 for all the application servers (see Section 5.3.4 below for details of the load balancer).

5.3.2 How much application work can memcached handle?

Now that we know *when* to invoke *SoftScale* (and thus, *when* to attempt to steal resources from the data tier), the next design question is: *how much* can we steal? The memcached servers are primarily responsible for providing data to the application work. Thus, we cannot overload memcached servers with too much application work. Figure 5.4(b) shows n^*_{mem}, the maximum number of application requests that a memcached server can handle simultaneously without violating the SLA. We see that n^*_{mem} depends on the overall system load, as should be expected. When the system load is low ($< 20\%$), each memcached server can handle almost half the work

capacity of an application server, whereas when the load is high ($\geq 80\%$), memcached servers cannot handle any application work.

In order to determine n^*_{mem}, we use the following methodology. For each initial load, we determine the number of application servers needed to handle this load, say k servers, using the arguments in Section 5.2.2. We then run a closed-loop experiment with k application servers and all 5 memcached servers, and we fix the number of simultaneous application requests in the system to $k \cdot n^*_{app}$. At this point, the application servers cannot handle any more simultaneous requests. We now increase the number of simultaneous requests in the system beyond $k \cdot n^*_{app}$, sending the additional requests to the memcached servers, and monitor T_{95}. Based on the initial load, each of the memcached servers will be able to handle some number of simultaneous application requests, n^*_{mem}, before T_{95} for the system rises above the SLA. Thus, we determine n^*_{mem} using a simple profiling approach (monitoring T_{95}).

5.3.3 Need for isolation

While we have successfully overloaded the functionality of the memcached servers, we have not eliminated interference between the memcached work and the application work at the memcached servers. One way of reducing interference is to "isolate" these two processes at the memcached servers, by partitioning the four cores of the memcached server between the memcached work and the application work. We achieve this core isolation by using the `taskset` command in Linux. A logical way of partitioning the cores is in a 2:2 ratio, with 2 cores dedicated to memcached work and 2 cores dedicated to application work. However, we find that the performance of *SoftScale* improves greatly if we *dynamically* adjust the partitioning based on total system load. For example, when the system load is extremely low, we can get away with restricting memcached to only one core at each memcached server and reserving the remaining three cores for application work in case of a load spike (1:3 partitioning). On the other hand, when the system load is very high, we need all four cores for memcached work (4:0 partitioning). Figure 5.5 shows n^*_{mem} for the memcached servers with dynamic isolation and without any isolation (same as Figure 5.4(b)). Note the four discrete horizontal levels for dynamic isolation. These refer to a 4-core partitioning between the memcached work and application work in the ratio of 1:3, 2:2, 3:1 and 4:0 respectively. We see that dynamic isolation greatly enhances the capacity of memcached servers to handle application work. Henceforth, when we use *SoftScale*, it will be implied that we are referring to *SoftScale* with dynamic isolation.

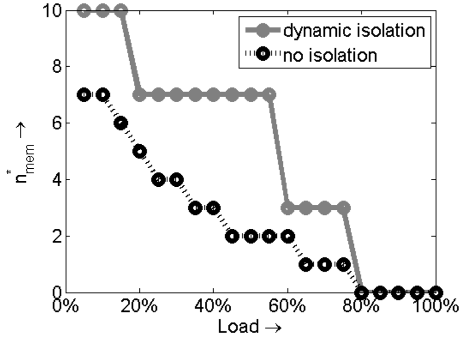


Figure 5.5: The figure illustrates enhancement in *SoftScale* using dynamic isolation.

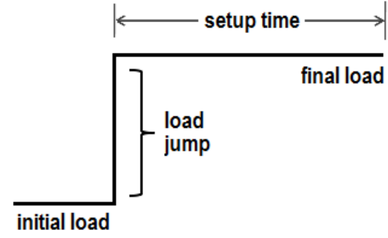


Figure 5.6: The figure illustrates the load jumps we use in our experiments. Note that we only evaluate the system during the setup time.

We obtain Figure 5.5 by repeating the same closed-loop experiments described at the end of Section 5.3.2 for each possible partitioning of the 4 cores among the memcached work and the application work. We then select the partitioning that gives us the highest n^*_{mem} value.

5.3.4 The SoftScale algorithm

We are now ready to describe our *SoftScale* algorithm, which is implemented in the load balancer, and is depicted in gray in Figure 5.2. We send application requests to the application servers, via Join-the-Shortest-Queue routing, as long as any server has less than n^*_{app} simultaneous requests. If all of the application servers have at least n^*_{app} requests, *SoftScale* is invoked. *SoftScale* sends any additional requests above the n^*_{app} requests to the memcached servers. The resource manager (see Figure 5.2) at each memcached server is responsible for invoking the software that will serve the incoming application requests. In our case, this software is the Apache web server with PHP support, which is invoked upon boot. The resource manager also isolates the application work from the memcached work. We limit the number of requests that we send to each memcached server to n^*_{mem}. Recall that n^*_{mem}, which is the optimal number of simultaneous application requests that a memcached server can handle, is not a constant, and in fact varies with load as specified in Section 5.3.3 and Figure 5.5. Note that $n^*_{mem} = 0$ if load is greater than or equal to 80% of peak load. Thus, *SoftScale* will not send application requests to the memcached servers if load is high. Once we have n^*_{mem} requests at all memcached servers, then we load balance additional requests among the application servers.

5.3.5 Analytical model for estimating *SoftScale*'s performance

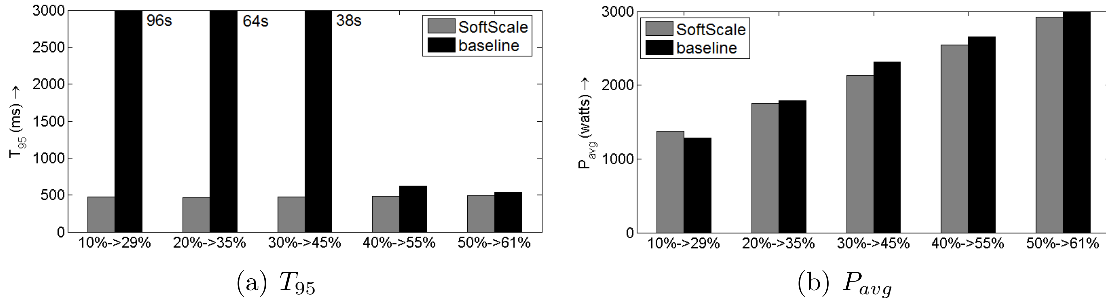
We now present a simple analytical model that allows us to estimate the range of load jumps that *SoftScale* can handle for a given multi-tier system. Let k_{app} and k_{mem} denote the total number of application servers and memcached servers in the system, respectively. If the current system load is x% of the peak load, where $0 \leq x \leq 100$, then the number of application servers on is roughly $k_{app} \cdot \frac{x}{100}$, assuming the application tier is dynamically scaled. Suppose that each memcached server can handle n^*_{mem} simultaneous application requests at load x%. Then, the total number of application requests that the memcached tier can handle is $k_{mem} \cdot n^*_{mem}$. Note that the number of simultaneous requests that the system can handle without *SoftScale* at load x% is $k_{app} \cdot \frac{x}{100} \cdot n^*_{app}$, where n^*_{app} is the number of simultaneous requests than an application server can handle. Thus, at x% load, the fraction of additional load that the system can handle with *SoftScale* is:

$$\text{Fraction of additional load that } \textit{SoftScale} \text{ can handle} \approx \frac{k_{mem} \cdot n^*_{mem}}{k_{app} \cdot \frac{x}{100} \cdot n^*_{app}} \tag{5.1}$$

Equation (5.1) suggests that the additional load that *SoftScale* can handle goes down as the system load (x%) increases, as expected (note that n^*_{mem} also drops with system load, as shown in Figure 5.5). As we will show in Sections 5.4.1 and 5.6, Equation (5.1) matches our experimental results for *SoftScale*'s performance. Thus, we can use Equation (5.1) to predict *SoftScale*'s performance for systems whose k_{app}, k_{mem}, n^*_{app}, or n^*_{mem} values are different from ours.

5.4 Results

We now evaluate the performance of *SoftScale* for a variety of load spikes. We start in Section 5.4.1, where we consider a range of *instantaneous* load jumps and characterize the space of jumps that *SoftScale* can handle. Then, in Section 5.4.2, we examine the performance of *SoftScale* under real-world load spikes. Finally, in Section 5.4.3, we examine the performance of *SoftScale* under load spikes that are caused by service outages or server failures. For all the experiments in this section we consider $t_{setup} = 5$ minutes, which is approximately the setup time for our servers. Later, in Section 5.5, we examine *SoftScale* under lower setup times.



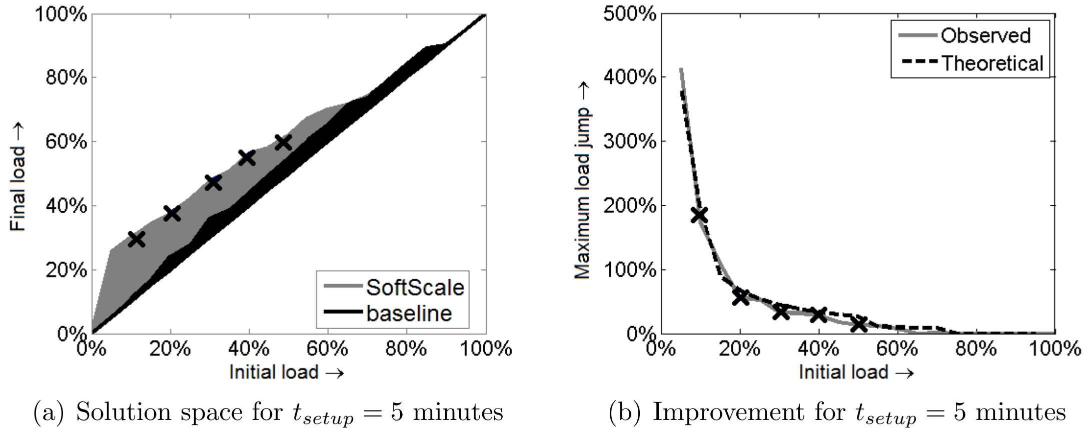
(a) T_{95} (b) P_{avg}

Figure 5.7: *SoftScale* meets $T_{95} \leq 500ms$ SLA without consuming any extra resources for a range of load jumps.

5.4.1 Characterizing the range of load jumps that *SoftScale* can handle

In this section we consider *instantaneous* jumps in load, as shown in Figure 5.6, and examine the system *only during the setup time*. We assume the system is properly provisioned for the initial load, and thus, is *under-provisioned* after the instantaneous load jump to the final load, during the setup time. Under *SoftScale*, although the application tier is under-provisioned during the setup time, we can use the memcached tier to compensate. By contrast, under the "baseline" architecture, we are limited to the capacity of the under-provisioned application tier. We compare *SoftScale* with the "baseline" architecture by examining the following metrics: T_{95}, the 95th percentile of response times during the 5 minute setup time, and P_{avg}, the average power consumed by the application servers and the memcached servers during the setup time. Note that P_{avg} is proportional to the amount of resources being used, and can thus be thought of as a proxy for operational costs. For a given load jump, if the system has $T_{95} \leq 500ms$, we say that it can "handle" the load jump.

Figure 5.7(a) shows the effect of *SoftScale* on T_{95} for specific load jumps. We choose these specific load jumps since they correspond to the maximum jump that *SoftScale* can handle at each of the initial loads. For example, if the initial load is 10% of the peak, then *SoftScale* can handle a maximum jump of 10% $\rightarrow$ 29%, where the load changes instantaneously from an initial load of 10% to a final load of 29%. We see that *SoftScale* provides huge benefits in T_{95}, as long as the final load is less than 50%. In particular, the T_{95} under *SoftScale* is less than 500ms for the 10% $\rightarrow$ 29% jump, as compared with 96s under the baseline. Likewise, *SoftScale* lowers T_{95} from 64s to less than 500ms for the 20% $\rightarrow$ 35%, and from 38s to less than 500ms for the 30% $\rightarrow$ 45% load jump. *SoftScale* provides these performance



(a) Solution space for $t_{setup} = 5$ minutes (b) Improvement for $t_{setup} = 5$ minutes

Figure 5.8: Full range of results for *SoftScale*. The crosses in the figures refer to the specific load jump cases shown in Figure 5.7.

improvements by opportunistically stealing resources from the memcached servers to handle the critical application work. When the load jumps from 40% $\rightarrow$ 55% and 50% $\rightarrow$ 61%, *SoftScale* still provides improvement in T_{95}, but these improvements are not as dramatic. This is because the memcached tier is optimally provisioned (see Section 5.2.2), and thus has very little spare capacity at high loads.

By contrast, the baseline (no *SoftScale*) would have to resort to significant over-provisioning to handle the load jumps. For example, for the 10% $\rightarrow$ 29% jump, the baseline would have to over-provision the application tier by about 190% to meet the SLA during the setup time. Clearly, this is a huge waste of resources.

Figure 5.7(b) plots P_{avg}, the average power consumed by the application servers and the memcached servers, for *SoftScale* and the baseline. We see that *SoftScale* does not consume any additional power as compared to baseline. This is because the total amount of work done by all servers under *SoftScale* and under baseline is about the same, for a given load level. Thus, P_{avg}, which is a proxy for operational costs, does not change significantly when using *SoftScale*.

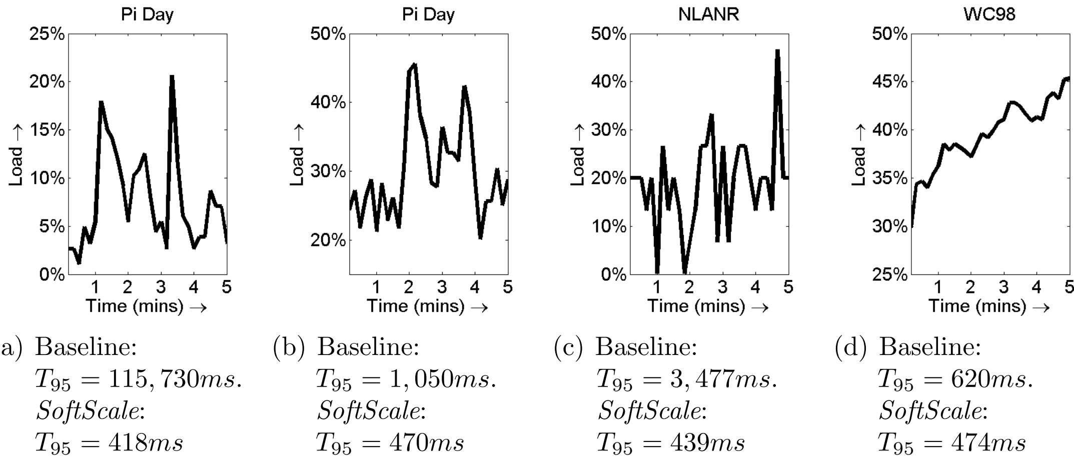
Figure 5.8 shows the full set of results for *SoftScale*. In Figure 5.8(a), the gray region shows the solution space, or regimes, of load jumps that *SoftScale* can handle without violating the 500ms SLA, while the black region shows the load jumps that the baseline can handle without violating the SLA. Note that *SoftScale*'s solution space is a superset of the baseline's solution space. The crosses in the figure refer to the specific load jump cases we showed in Figure 5.7, namely the maximum load jumps that *SoftScale* can handle for each of the initial loads.

Since the system is optimally provisioned (see Section 5.2), the baseline cannot handle any significant load jumps. In particular, when the initial load is either too low or too high, the baseline cannot handle any load jumps. However, because of the inherent elasticity in the system, the baseline can handle some small load jumps when the initial load is moderate. For example, when the initial load is 20%, the black region indicates that baseline can handle a maximum jump of 20% → 24%.

By contrast, *SoftScale* can handle a much larger range of load jumps as compared to the baseline. For example, when the initial load is 20%, the gray region indicates that *SoftScale* can handle a maximum jump of 20% → 35%.

In Figure 5.8(b), we plot the maximum load jump (in %) that *SoftScale* can handle for each initial load using the solid gray line. Again, the crosses in the figure refer to the specific load jump cases we showed in Figure 5.7. For example, the first cross from the left corresponds to the 10% → 29% load jump, which amounts to a 190% jump in load. The dashed line shows our estimates for the maximum load jump that *SoftScale* can handle, given by Equation (5.1) (with a few extra % due to the elasticity in the system). We see that our estimates match our implementation results. As expected, Figure 5.8(b) shows that *SoftScale* can handle huge jumps when the initial load is low, but can only handle moderate load jumps when the initial load is high.

In our experimental evaluation above, we only invoke *SoftScale* during a load spike. However, in theory, one could use *SoftScale* all the time, that is, even when there is no load spike. For example, in our testbed shown in Figure 5.2, we could leverage the spare capacity in the caching tier during low loads to reduce the number of required application tier servers. We can also leverage the spare capacity in the caching tier servers by physically colocating the application tier and the caching tier on the same servers (see, for example, [47, 126]). The advantage of using *SoftScale* perpetually is that it reduces system power consumption. For example, for a 35% load, we provision $\lceil \frac{262}{37.5} \rceil = 7$ application servers. However, from Figure 5.7(a), we know that *SoftScale* can handle a 20% → 35% load jump. Thus, by using *SoftScale* perpetually, we only need to provision enough application servers for a 20% load, which is 4 application servers. We thus save 3 application servers' worth of power, which is roughly $140 \times 3 = 420$ watts, since the idle power of an application server is 140 watts. Given that the baseline system consumes about 1,790 watts of power when the load is 35% (see Figure 5.7(b)), using *SoftScale* perpetually reduces power consumption by about $\frac{420}{1790} \times 100 = 23\%$. Similarly, for loads of 45% and 55%, using *SoftScale* perpetually reduces power consumption by about 18% and 15% respectively. The power savings decrease as the load increases because of the reduction



(a) Baseline: $T_{95} = 115,730ms$. *SoftScale*: $T_{95} = 418ms$

(b) Baseline: $T_{95} = 1,050ms$. *SoftScale*: $T_{95} = 470ms$

(c) Baseline: $T_{95} = 3,477ms$. *SoftScale*: $T_{95} = 439ms$

(d) Baseline: $T_{95} = 620ms$. *SoftScale*: $T_{95} = 474ms$

Figure 5.9: Real-world trace snippets used for our experiments.

in spare capacity in the caching tier. The obvious disadvantage of using *SoftScale* perpetually is that the system can no longer handle abrupt changes in load, and is thus vulnerable to load spikes. In summary, there is a tradeoff between power savings and robustness to load spikes when deciding on how to use *SoftScale*. Since our focus in this chapter is on making the system robust to load spikes, we only use *SoftScale* when there is an abrupt increase in load.

5.4.2 Spikes in real-world traces

In addition to evaluating *SoftScale* under instantaneous load jumps (as in Section 5.4.1), we also evaluate *SoftScale* under the real-world traces, Pi Day [18], NLANR [144], and WC98 [20], shown in Figure 5.9. We re-scale each trace so that the peak load corresponds to 750 req/s, and then consider five minute (t_{setup}) snippets that highlight load spikes. The load numbers in Figure 5.9 correspond to the post-scaled traces. We assume the system is well provisioned at time $t = 0$, and then examine the system performance for the next five minutes, during which additional capacity is being brought online.

Although the initial load ranges from 5% to 30% across the different traces, *SoftScale* achieves a T_{95} of less than 500ms for all cases (see Figures 5.9(a) to 5.9(d)). By contrast, the baseline results in a T_{95} of over 115s in Figure 5.9(a), where the load quadruples from 5% to 20%. In Figure 5.9(b), where the load roughly doubles from 25% to 46%, the T_{95} under the baseline is just over a second, in contrast to *SoftScale*'s 470ms. The superiority of *SoftScale* over the baseline for the trace in

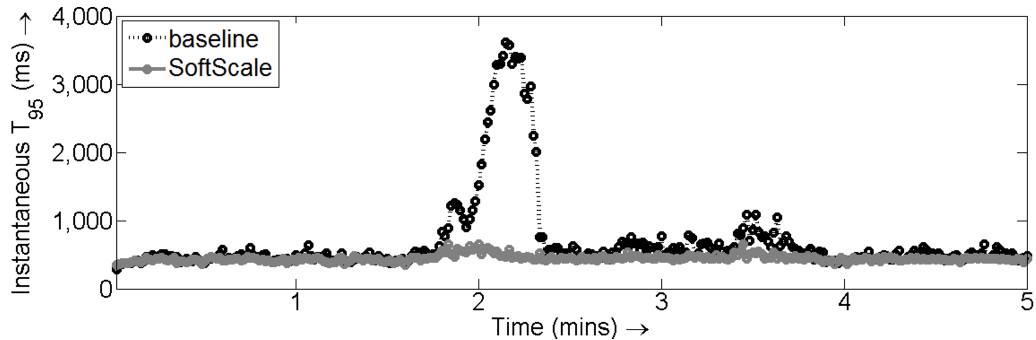


Figure 5.10: The plot illustrates the superiority of *SoftScale* over the baseline for the Pi Day [18] trace in Figure 5.9(b).

Figure 5.9(b) is further illustrated in Figure 5.10, which depicts the instantaneous T_{95} (collected every second) over the trace.

5.4.3 Spikes created by server faults

Thus far, we considered the case where load spikes are caused by a sudden increase in request rate. However, load spikes can also result because of a sudden drop in capacity. Service outages [147] and server failures [162] are common causes for a sudden (and unpredictable) drop in capacity. *SoftScale* is useful regardless of the cause of load spikes since it is invoked when the number of requests at a server increases (see Section 5.3.1). We now illustrate the fault-tolerance benefits of *SoftScale*.

Consider a system that is well provisioned to handle 30% initial load. Suppose a failure takes down half of the provisioned capacity, resulting in a system that can now only handle 15% load. We refer to this as a 30% $\rightarrow$ 15% capacity drop. Figure 5.11(a) shows our experimental results for instantaneous T_{95} (collected every second) under a 30% $\rightarrow$ 15% capacity drop, which is triggered at the 10s mark. Apache's load balancer is very quick to recognize that some of the application servers are offline, and thus stops sending additional requests to them. In Figure 5.11(a), while *SoftScale* successfully handles the capacity drop, the baseline completely falls apart. The power consumption for *SoftScale* and the baseline are about the same, and are thus omitted due to lack of space.

Figure 5.11(b) shows our experimental results for instantaneous T_{95} under a very severe 50% $\rightarrow$ 20% capacity drop, which is produced by taking down 6 of the 10 application servers at the 10s mark. This time, we see that instantaneous T_{95} rises sharply for both *SoftScale* and the baseline. However, the rate at which instantaneous

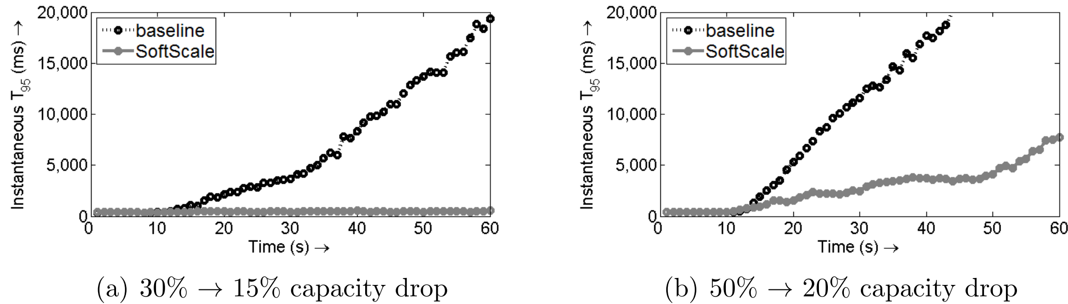


Figure 5.11: *SoftScale* provides significant benefits even when load spikes are caused by a sudden drop in capacity. In the figures above, we drop capacity at the 10s mark.

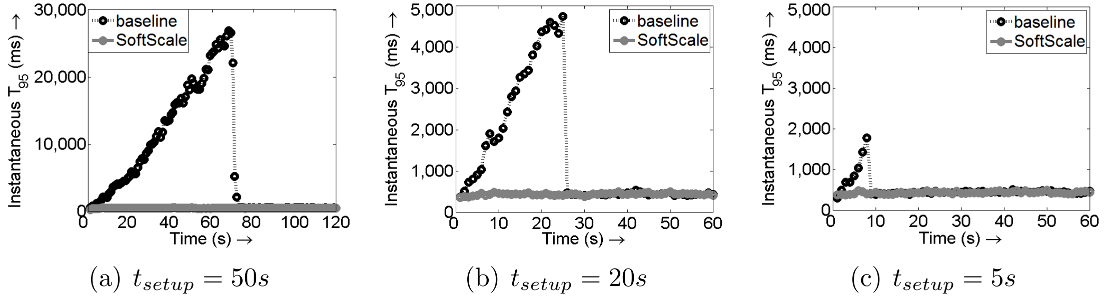
T_{95} increases under *SoftScale* is significantly lower than that under the baseline. Thus, we conclude that *SoftScale* is useful even when load spikes are caused by a sudden drop in capacity.

5.5 Lower Setup Times

While production servers today are only equipped with "off" states that necessitate a huge setup time, future servers may support sleep states, which can lower setup times considerably. Further, with virtualization, the setup time required to bring up additional capacity (in the form of virtual machines) might also go down. In this section we analyze *SoftScale* for the case of lower setup times by tweaking our experimental testbed as discussed in Section 2.3.2. Intuitively, for low setup times, one might expect that *SoftScale* is not needed since instantaneous T_{95} should not rise too much during the setup time. This turns out to be false.

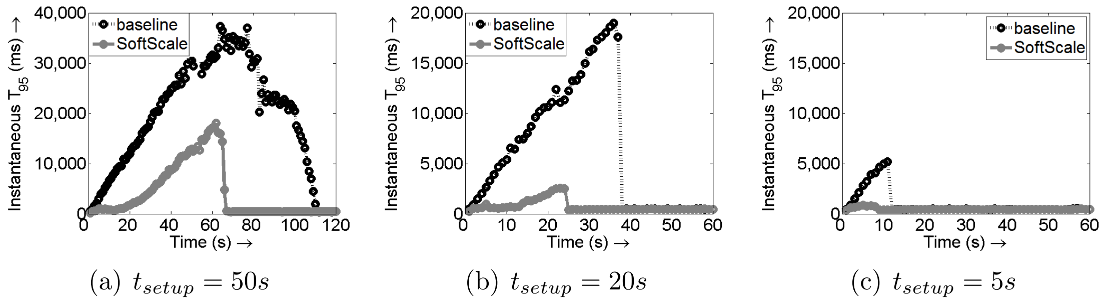
Figure 5.12 shows our experimental results for instantaneous T_{95} under the 15% → 30% load jump, for a range of t_{setup} values. We change the scale for Figure 5.12(a) to fully capture the effect of the 50s setup time. Recall from Figure 5.8(a) that *SoftScale* can handle the 15% → 30% load jump, even if t_{setup} = 5 minutes. Thus, it is not surprising that *SoftScale* can handle the 15% → 30% load jump for t_{setup} = 50s, 20s and 5s in Figure 5.12.

By contrast, the instantaneous T_{95} for the baseline quickly grows and exceeds the 500ms SLA during the entire setup time duration, even for the $t_{setup} = 5s$ case. However, the instantaneous T_{95} values for the baseline are not too high under lower setup times. This is because *when the setup time is low, the overload period is very*



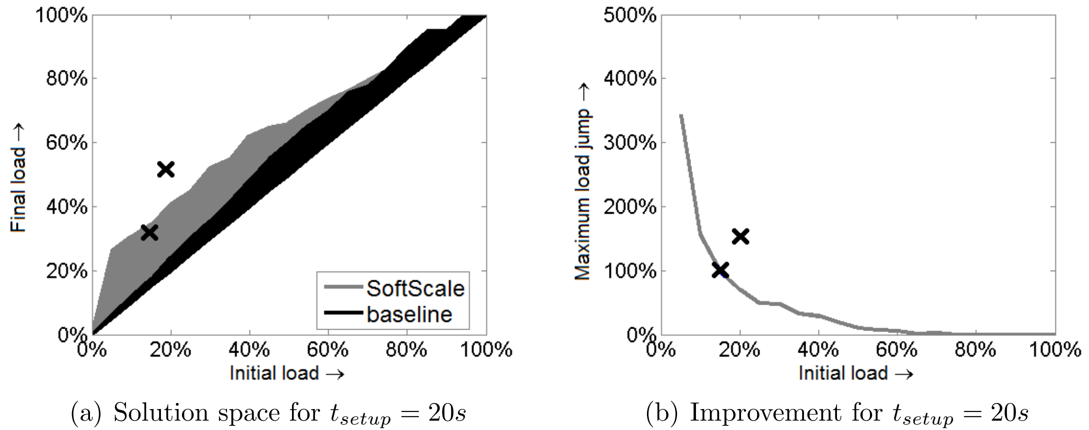
(a) $t_{setup} = 50s$ (b) $t_{setup} = 20s$ (c) $t_{setup} = 5s$

Figure 5.12: Effect of t_{setup} on instantaneous T_{95} for a 15% → 30% jump in load.



(a) $t_{setup} = 50s$ (b) $t_{setup} = 20s$ (c) $t_{setup} = 5s$

Figure 5.13: Effect of t_{setup} on instantaneous T_{95} for a 20% → 50% jump in load.



(a) Solution space for $t_{setup} = 20s$ (b) Improvement for $t_{setup} = 20s$

Figure 5.14: Full range of results for *SoftScale* under $t_{setup} = 20s$. The crosses in the figures refer to the specific load jump cases shown in Figures 5.12 (15% → 30% load jump) and 5.13 (20% → 50% load jump).

short. Observe that instantaneous T_{95} does not drop immediately after the setup time because of the backlog in requests created during the setup time.

Figure 5.13 shows our experimental results for instantaneous T_{95} under the 20% → 50% load jump. Recall from Figure 5.8(a) that *SoftScale cannot* handle the 20% → 50% load jump when $t_{setup} = 5$ minutes. In Figure 5.13, we see that instantaneous T_{95} rises sharply during the setup time for both *SoftScale* and the baseline. However, the rate at which instantaneous T_{95} increases under *SoftScale* is at most half that under the baseline.

Figure 5.14 shows the full set of results for *SoftScale* for the case of $t_{setup} = 20s$. In Figure 5.14(a), we show the solution space of load jumps that *SoftScale* and the baseline can handle without violating the 500ms T_{95} SLA (over the 20s setup time). The crosses in the figure refer to the specific load jump cases we showed in Figures 5.12 and 5.13. We see that *SoftScale* can handle a much larger range of load jumps (gray region) as compared to the baseline (black region), just as we observed in Figure 5.8(a) for $t_{setup} = 5$ minutes. In Figure 5.14(b), we plot the maximum load jump (in %) that *SoftScale* can handle for each initial load. Again, as expected, *SoftScale* can handle huge jumps when the initial load is low, but can only handle moderate load jumps when the initial load is high.

It is very interesting to note that the performance degradation caused by load spikes for the baseline case does not go away even when the setup time is really low. Thus, *there is a need for* SoftScale *even under low setup times.* Comparing Figures 5.8 and 5.14, we see that the range of load jumps that the baseline (and *SoftScale*) can handle increases only slightly under the much lower setup time of 20s. The reason that this increase is so small is that most of the "damage" to T_{95} has already occurred after only a few seconds.

5.6 Future Architectures

In our implementation testbed (see Section 5.2), we use 4-core servers for the memcached tier. In the near future, it is likely that 4-core processors will be replaced by 8 (or more) core processors, even though their memory capacity is unlikely to increase significantly [101]. Thus, we would still need just as many memcached servers. On the other hand, data replication needs may require additional memcached servers. In either case, the memcached tier will now have more spare compute capacity that can be exploited by the application tier via *SoftScale*. In this section we investigate the performance of *SoftScale* for the case where we have 8-core memcached servers.

Figure 5.15 shows n^*_{mem}, the optimal number of application requests that a memcached server can handle simultaneously without violating the 500ms SLA, for 8-core and 4-core memcached servers. We see that using 8-cores allows us to put a lot more

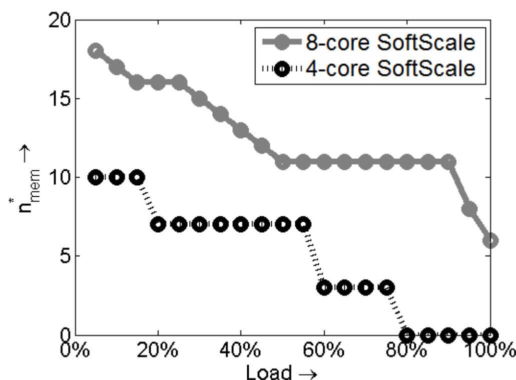


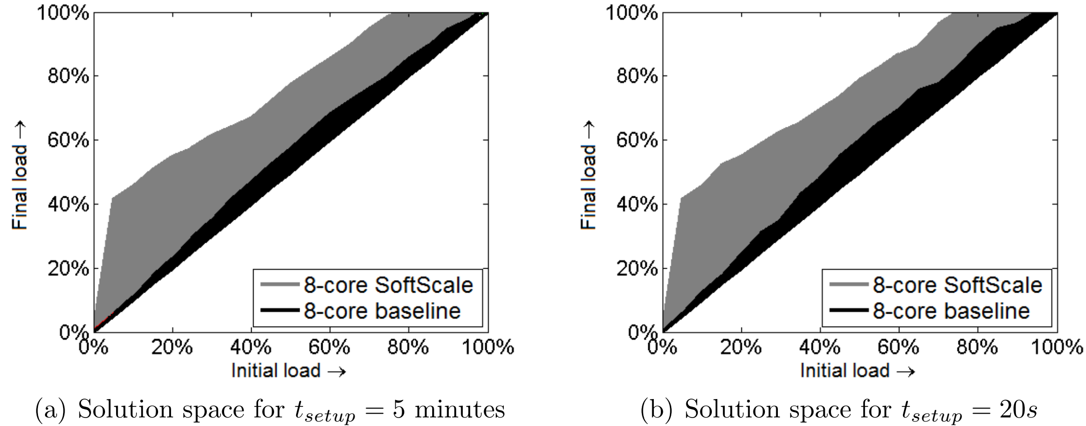
Figure 5.15: Using 8-core memcached servers significantly enhances *SoftScale*'s ability to handle load jumps.

application work on the memcached servers. Thus, *SoftScale* should be able to handle much higher load jumps with 8-core memcached servers.

Figure 5.16(a) shows the full set of results for *SoftScale* and the baseline, both with 8-core memcached servers, for the case of t_{setup} = 5 minutes. We see that *SoftScale* with 8-core memcached servers can handle a significantly larger range of load jumps. For example, *SoftScale* can handle a 10% → 50% load jump as compared to the maximum jump of 10% → 29% using 4-core memcached servers, as was shown in Figure 5.8(a). Further, *SoftScale* can now handle load jumps even when the load is as high as 80%, since the memcached work requires at most 4 cores at peak load (see Section 5.2.2), still leaving 4 cores at each memcached server for application work. We also estimated the maximum load jump that *SoftScale* can handle via Equation (5.1), and found that our estimates match our implementation results. Figure 5.16(b) shows the full set of results for *SoftScale* for the case of $t_{setup} = 20s$. These results are very similar to those in Figure 5.16(a). Thus, even though there is a cost (monetary cost and increased power consumption) involved in switching to 8-core memcached servers, it might make sense to deploy these servers for the memcached tier to handle severe load spikes using *SoftScale*.

5.7 Prior Work

There is a lot of prior work that deals with dynamic server provisioning (see Section 2.2). However, most of the approaches to dynamic server provisioning, including our approach, *AutoScale* (see Chapter 4), cannot handle load spikes. This claim was also verified by other authors [38].



(a) Solution space for $t_{setup} = 5$ minutes (b) Solution space for $t_{setup} = 20s$

Figure 5.16: Full range of results for *SoftScale* with 8-core memcached servers under (a) $t_{setup} = 5$ minutes and (b) $t_{setup} = 20s$. We see that 8-core memcached servers provide huge benefits for *SoftScale* regardless of the setup time.

There has been some prior work specifically dealing with load spikes [38, 110]. Chandra et al. [38] show that existing dynamic server provisioning policies are not good at handling flash crowds in an internet data center. In order to handle flash crowds, the authors advocate either having spare servers that are always available (over-provisioning), or finding a way to lower setup times. However, as our work shows (see Figures 5.12(c) and 5.13(c)), even a 5s setup time can result in severe SLA violations. Further, by using *SoftScale*, we do not have to pay for any additional resources, which is not the case when over-provisioning via spare servers. Lassettre et al. [110] propose a short-term forecasting approach to handle load spikes for a multi-tier system with a setup time of 30s. While [110] is very effective at handling load spikes that gradually build over time, it is not well suited for the instantaneous load spikes we consider in this chapter since the forecasting in [110] itself requires at least 10s, and we have shown that even a 5s setup time is detrimental. Observe that *SoftScale* is actually complementary to the above approaches, and can be used in conjunction with them.

There has also been recent work looking at data spikes, where a particular web object becomes extremely popular. Data spikes can be handled by caching or replication techniques (see, for example [172]), and are not the focus of our work.

To handle load spikes for small websites with only static content, a possible solution is to host their content on a cloud computing platform. These platforms are able to handle load spikes by over-provisioning more economically since they host multiple

websites, and load spikes on individual websites are often not correlated (statistical multiplexing) [54]. For multi-tiered cloud computing environments, *SoftScale* can be used in conjunction with statistical multiplexing.

Finally, there is also a lot of prior work [175, 8, 185, 42] that deals with managing overload conditions by allowing for performance degradation. Some of the popular techniques that have been used to regulate performance degradation include admission control and request prioritization. By contrast, *SoftScale* handles load spikes without any performance degradation, provided the load is not high. If the load is high, *SoftScale* can be coupled with techniques like those in [175, 8, 185, 42] to minimize the damage caused by load spikes.

5.8 Chapter Summary

In this chapter we address the challenges presented by load spikes, which are all too common in today's data centers [44, 145, 113, 88, 20, 147, 162]. Our results in Figures 5.12 and 5.13 show that ignoring load spikes can result in severe SLA violations, even if it takes only 5 seconds of setup time to bring capacity online. The obvious solution of over-provisioning resources is quite expensive since load spikes are often unpredictable.

We propose *SoftScale*, an approach to handling load spikes in multi-tier data centers without consuming any extra resources. In multi-tier data centers, the application tier is typically stateless, and can be dynamically provisioned, whereas the data tier is stateful, and is always left on. *SoftScale* works by opportunistically stealing resources from the data tier to alleviate the overload at the application tier during the setup time needed to bring additional application tier capacity online. Since tiers in a data center are typically carefully provisioned for peak load, *SoftScale* must steal from the data tier without hurting overall performance. *SoftScale* does this by first determining how much spare capacity can be stolen from the data tier without violating SLAs at different load levels, and then dynamically isolating the application work and the data delivery work at the data tier to avoid interference.

Our implementation results on a 28-server testbed demonstrate that *SoftScale* can handle various load spikes for setup times ranging from 5 seconds to 5 minutes (see Figures 5.8 and 5.14). Specifically, *SoftScale* can handle instantaneous load jumps ranging from 5% $\rightarrow$ 25% to 50% $\rightarrow$ 61%, even when the setup time is 5 minutes. *SoftScale* works extremely well for real-world load spikes (see Figure 5.9), and significantly improves performance (typically a 2X – 100X factor improvement)

when compared to the baseline. Even greater benefits are likely possible for future many-core servers (see Figure 5.16).

While our implementation testbed mimics a web site of the type seen in Facebook or Amazon with an application tier and a memcached tier, we believe *SoftScale* will also be applicable when the memcached tier is replaced by any other data tier. Since the data tier is stateful, there will always be a subset of servers that will not be turned off. Thus, *SoftScale* can leverage these servers to alleviate the bottleneck at the application tier during load spikes.

In this work we consider a data center architecture where the stateless and stateful tiers are physically separate. This division is preferable because it makes it easier to scale and manage the individual tiers [52, 163, 173]. One might argue that colocating the stateless and stateful tiers leads to better resource sharing. Unfortunately, colocating different services on the same physical servers is not an easy task [105, 141], and is a research question in itself [28, 80]. For *SoftScale*, we only have to deal with colocation (addressed via core isolation) during the setup time. However, if we have a data center with colocated tiers, we will have to deal with the consequences of colocation at all times. Importantly, colocated tiers significantly reduce the potential for dynamic provisioning, because all servers in a colocated data center are stateful.

SoftScale relies on the fact that the data tier is always on. This holds true for most existing systems [172, 34], even if there is a significant drop in load [24]. In fact, prior work on scaling the data tier has been limited to adjusting the replication factor of the storage system [14, 170]. Nevertheless, we view *SoftScale* as an alternative to scaling the data tier. In the next chapter, we explore dynamic provisioning of the data tier.

Chapter 6

CacheScale: Dynamic Provisioning of the Caching Tier

In this chapter we consider dynamic provisioning of the caching tier. The servers in a caching tier are typically [128] provisioned with massive amounts of expensive and power-hungry DRAM. Thus, careful dynamic provisioning of the caching tier can lead to huge savings in cost and power. In fact, our results in this chapter show that a 2x drop in load can result in up to 90% savings in the caching tier capacity. Unfortunately, scaling the caching tier is very challenging because of the (temporary) data unavailability that follows the addition or removal of a caching server, which in turn leads to high response times (see Observation 3.8). This is illustrated in our experimental results in Figure 3.15. To the best of our knowledge, there has been no prior work on dynamic provisioning of the caching tier.

We present *CacheScale*, a novel dynamic provisioning approach that meets response time SLAs while scaling the caching tier capacity. *CacheScale* ensures that scaling the caching tier does not significantly impact hit rate by proactively moving hot data items. Importantly, *CacheScale* does not require access to the elusive least-recently-used list of cached items for moving hot data. This makes it very easy to deploy *CacheScale* on any caching tier.

We introduce the problem and discuss the scope of this chapter in Section 6.1. We describe our experimental setup in Section 6.2. We then analyze the potential savings in the caching tier size that can be achieved by dynamic provisioning in Section 6.3. In order to realize these savings, we propose *CacheScale*. We discuss the design and implementation of *CacheScale*, and experimentally evaluate its benefits in Section 6.4. We conclude with a summary of this chapter in Section 6.5.

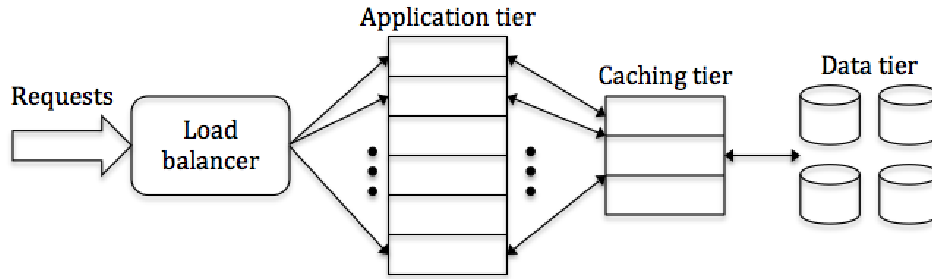


Figure 6.1: Multi-tier cloud service.

6.1 Introduction

We consider a multi-tier web service hosted in a cloud computing environment, as shown in Figure 6.1. Each "server" in this case is a virtual machine, such as those provided by Amazon EC2 [13]. The front-end application tier consists of a set of stateless application servers that process requests using data from the back-end. The back-end data tier consists of a persistent storage system, such as a database. To alleviate load at the back-end, a stateful, but non-persistent, distributed caching tier is often used [59, 128] to cache data or partial results.

With cloud computing, web service providers have the ability to dynamically scale their computing infrastructures to match demand. Further, because cloud resources are often priced per-use, web service providers have a monetary incentive to minimize the number of resources consumed while still meeting the SLAs of the service. Each tier must be treated differently when scaling. The application tier is the easiest tier to scale because it is stateless [107, 40]. In contrast, because web services often impose stringent data availability requirements, options for scaling the data tier are typically limited to adjusting the replication factor of the storage system [14, 170].

Note that the tiers are typically of different sizes, with the servers in the application tier often outnumbering servers in the caching tier at a ratio somewhere around 4:1 [55]. While this fact might initially suggest that scaling the caching tier may not lead to significant savings, note that DRAM is an expensive resource. Further, caching tier servers are often equipped with huge amounts of DRAM [128]. As an example, if we were to instantiate a testbed of 20 application servers and 5 cache servers on EC2 [13], our caching tier would represent 41%[1] of the operational cost, despite the fact that our caching tier only comprises 5 servers, and thus represents 20% of

[1] Assuming application servers cost $0.165/hr/high-cpu instance [13] and cache servers cost $0.45/hr/high-memory instance [13].

the total capacity. Further, at times of lower utilization, after the application tier has been scaled down, an unscaled caching tier becomes a more significant fraction of the service's operational cost. In the above example, if load were to drop by a factor of 4, and we were to scale down our application tier from 20 servers to 5 servers, our caching tier would then represent 73%[1] of the operational cost. Thus, scaling down the number of cache instances as the load decreases could provide significant cost benefits. In a non-cloud environment (physical infrastructure), these cost benefits would translate to power savings obtained by turning off unneeded cache servers.

In order to study dynamic provisioning of the caching tier, we limit our scope to (predictable) long-term variations in load. In particular, we ignore short-term fluctuations in demand, and thus avoid setup costs. Given this scope, there are at least a couple of technical challenges that we must address. First, as the caching tier scales down, the amount of cached data decreases. Will the resulting cache misses overwhelm the database? Second, how should the caching tier be managed so that "hot" data is preserved when cache instances are removed and distributed when cache instances are added?

The *key insight* in answering the first question is that when the overall load drops, we can afford a higher fraction of requests going to the database; hence, we can tolerate a lower cache hit rate, and this lower cache hit rate translates to a vast reduction in the amount of data cached. To correctly size the caching tier, we work backwards – when the load drops, we determine the minimum cache hit rate needed to ensure that the response time SLA is met (thereby limiting how many requests go to the database). We then calculate the cache size that would provide that hit rate. A key parameter that determines the degree to which the cache capacity may be reduced without overwhelming the database is the distribution of requests. If the requested data items are uniformly distributed, the degree to which cache capacity may be reduced is small. However, many studies have shown that web requests follow a very skewed distribution (often modeled as a Zipf distribution [31, 12]). In such cases, we show that significant savings are possible. In Figure 6.2, we see that a relatively small change in the required hit rate – from 0.95 to 0.8 – may result in a substantial reduction in the cache size needed under a Zipf distribution (79% to 34% of the data), but only a modest reduction (95% to 80%) if the distribution is Uniform.

In answering the second question regarding cache management, we first rely on consistent hashing techniques to ensure that excessive data migration is not needed as instances are added and removed from the caching tier. However, even with consistent hashing, naïvely adding or removing a cache instance can cause significant

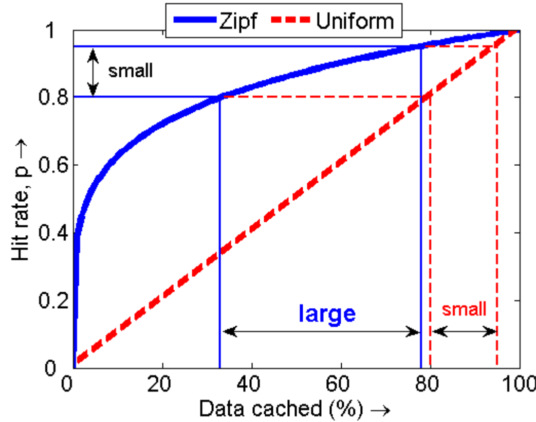


Figure 6.2: A small decrease in cache hit rate can lead to a large decrease in the amount of cached data.

performance problems, because the hot data is distributed over all instances. In Section 6.4, we present *CacheScale*, a simple approach that reduces the number of cache misses during periods of scaling by redistributing cached items.

Our contributions in this chapter are:

- We demonstrate that the caching tier can be effectively scaled with load without violating response time SLAs.
- We develop a model to calculate (i) the required hit rate as a function of load, and (ii) the resulting savings in the size of the caching tier obtained by scaling down during low load (see Section 6.3).
- We validate our theoretical results via implementation on a 28-server testbed (see Section 6.3.4).
- We propose a *simple* cache management technique, *CacheScale*, to redistribute cached items as the caching tier scales up and down, significantly reducing transient cache misses.

6.2 Experimental Setup

We experiment with a testbed of 28 servers which are divided into multiple tiers as in Figure 6.1. We employ one of these servers as the load generator running httperf [138]. Another server is used as the load balancer running Apache HTTP Server, which distributes PHP requests from the load generator to 20 application

servers. The application servers (Intel Xeon E5520 processor-based) parse the incoming PHP requests and collect the required data from the caching tier and the database. In our experimental setup, we use a distributed cache, memcached [59]. The caching tier comprises 20 memcached instances, each with up to 10GB of memory for caching, hosted on 5 servers (Intel Xeon X5650 processor-based). The database comprises a server (Intel Xeon E5520 processor-based) with 5 disks running an Oracle BerkeleyDB [146] database with a billion key-value pairs (250GB).

We design a key-value workload to model realistic multi-tier applications such as the social networking site, Facebook, or e-commerce sites like Amazon [48]. Each workload (HTTP) request is a PHP script that runs on the application server, and consists of 20 independent key-value fetches. The fetched keys follow a Zipf [143] distribution. Each of the 20 key-value fetches either hits in the memcached, or, if it misses, goes to the database. If a key-value fetch hits in the memcached, its response time is $T_{mem} = 0.3ms$, which is the time to retrieve a key-value pair from memcached. If a key-value fetch goes to the database, its response time is on the order of 8ms, depending on the contention at the database. In this chapter our SLA requirement is that the average response time for the entire request (collection of 20 key-value fetches), which we refer to as T_{avg}, should be no more than $T_{SLA} = 100ms$. Thus, we require:

$$T_{avg} \leq T_{SLA} = 100ms. \tag{6.1}$$

6.3 Assessment of Cache Savings

In this section we investigate the potential savings that can be achieved by scaling down the caching tier, without violating the response time SLA. We first describe our theoretical framework that allows us to estimate these savings, and then report our experimental results, which validate our estimates.

6.3.1 Popularity distribution

Many websites report that their data popularity distribution is far from being Uniform or Random, and is often very skewed. That is, a small subset of the entire data set is responsible for most of the traffic [164, 31, 12, 56]. Thus, the obvious caching solution is to cache this small subset of data in the caching tier. In fact, given the popularity distribution, we can estimate the amount of data to be cached to achieve a given hit rate.

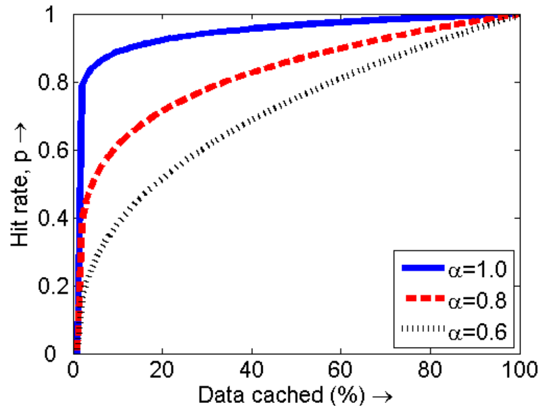


Figure 6.3: The Zipf popularity distribution.

Researchers often use a Zipf distribution to represent the website popularity distribution [164, 31, 12]. Mathematically, if the data items are sorted in decreasing order of popularity, the Zipf distribution states that the probability of seeing a request for data item i, $Pr\{i\}$, is:

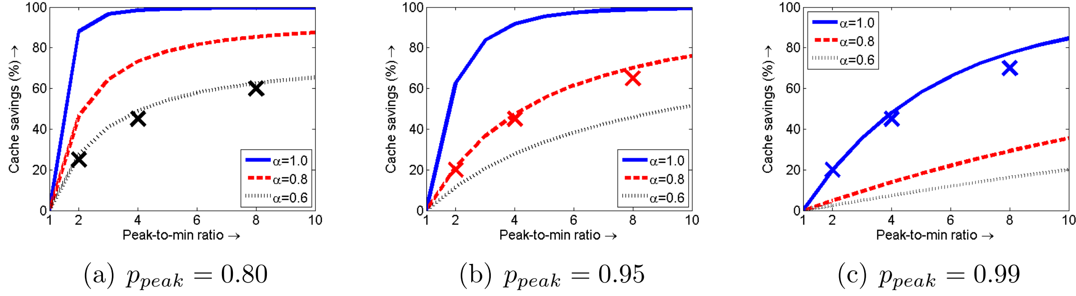
$$Pr\{i\} = \frac{C}{i^{\alpha}}, \tag{6.2}$$

where α is a parameter of the distribution, and C is a normalization constant. For real-world website traffic, α is typically between $0.6 - 1.0$ [164, 31]. A higher value of α corresponds to a more skewed distribution. We can now use Equation (6.2) to estimate the amount of data to be cached to achieve a given hit rate, p. Figure 6.3 shows these results for a Zipf popularity distribution with a range of α values. When α is high, say $\alpha = 1.0$, we can achieve a hit rate of $p = 0.8$ by caching only 2% of the data. However, when α is low, say $\alpha = 0.6$, we need to cache almost 60% of the data to achieve the same $p = 0.8$ hit rate. For a Uniform distribution ($\alpha = 0$), we would have to cache 80% of the data to achieve $p = 0.8$ hit rate.

6.3.2 Theoretical model

In Figure 6.3, we saw the relationship between the hit rate, p, and the amount of data cached. We now investigate the relationship between a given response time SLA, T_{SLA}, and the *minimum* required hit rate, p. This allows us to calculate the amount of cache required to meet T_{SLA}. The relationship between T_{SLA} and p is given by:

$$T_{avg} = 20\left(p \cdot T_{mem} + (1-p) \cdot T_{DB}\right) \leq T_{SLA} = 100ms, \tag{6.3}$$



(a) $p_{peak} = 0.80$ (b) $p_{peak} = 0.95$ (c) $p_{peak} = 0.99$

Figure 6.4: Theoretical savings in cache for a given peak-to-min ratio. Crosses indicate experimental results.

where $T_{mem} = 0.3ms$ is the latency for fetching a key-value pair from memcached, and T_{DB} is the latency for fetching a value from the database. Here, we use the fact that each request in our system is composed of 20 individual key-value fetches.

Note that T_{DB} in Equation (6.3) is not a constant, and depends on the request rate into the database, λ_{DB}. As λ_{DB} increases, so does T_{DB}. We model this relationship using an $M/M/1$ queueing model with load-dependent service rates [103]. This gives us T_{DB} as a function of λ_{DB}.

If λ denotes the total request rate into the system, then λ_{DB} is:

$$\lambda_{DB} = 20 \cdot \lambda \cdot (1 - p). \tag{6.4}$$

If the request rate into the database, λ_{DB}, is too high, then T_{DB} grows to infinity. Thus, it is important to limit the request rate to the database by keeping $(1-p)$ small (the hit rate, p, should therefore be high). Also, since T_{mem} is orders of magnitude smaller than T_{DB}, it is desirable to keep p high so as to make T_{avg} less than T_{SLA}. While both these goals point towards making p as *high* as possible, our objective in this chapter is to find the *lowest possible* p which still meets the desired T_{SLA} constraint (this is because low p implies low cache size from Figure 6.3).

Given the request rate into the system, we can solve Equation (6.3) for p, which in turn is used to calculate the desired cache size from Figure 6.3.

6.3.3 Theoretical results

Web services often exhibit huge variations in their request rates, mostly due to the diurnal/periodic nature of traffic. Assuming that the system is well provisioned to meet T_{SLA} for peak request rate, we are interested in the potential for decreasing

the cache size when the request rate decreases. We refer to the ratio between the peak request rate and the lower request rate as the peak-to-min ratio. We pick a range of peak-to-min ratios $(1 - 10)$ and use our theoretical model to calculate the possible reduction in cache size. We also show results across different values of α: 1.0 (heavily skewed), 0.8, and 0.6 (less skewed). Lastly, we vary the hit rate at the peak request rate: $p_{peak} = 0.80$ (Figure 6.4(a)), $p_{peak} = 0.95$ (Figure 6.4(b)), and $p_{peak} = 0.99$ (Figure 6.4(c)). We choose these parameter values based on recent studies [164, 27, 31, 12].

Figure 6.4 indicates that *significant cache reductions (up to 90%) are possible even for a 2:1 peak-to-min ratio.* In general, *the potential savings in cache size are higher when α is high.* This is because a higher value of α indicates a more skewed popularity distribution, which allows for more aggressive reduction in cache size (see Figures 6.2 and 6.3). Also, *the observed potential savings in cache size are higher when p_{peak} is low.* This can be explained as follows. If the request rate into the system, λ, drops by a factor of (say) 2, then Equation (6.4) tells us that the miss-rate, $(1 - p)$, can increase by a factor of 2 while maintaining the same request rate into the database, λ_{DB}. For example, when the peak hit rate is low, say, $p_{peak} = 0.8$, the miss-rate $(1 - p_{peak}) = 0.2$. If the request rate now drops by a factor of 2, our final hit rate can be as low as $p = 0.6$, since the miss-rate, $(1 - p)$, can now be as high as $2 \cdot (1 - p_{peak}) = 0.4$. However, when the peak hit rate is $p_{peak} = 0.99$ and the request rate drops by a factor of 2, our final hit rate can only be as low as $p = 0.98$. This implies that when the peak hit rate is low, we can afford a larger drop in hit rate, which in turn implies larger cache savings via Figure 6.3.

These large cache savings translate to huge cash savings. Consider an application hosted on EC2 [13] requiring a 4:1 ratio between application instances (costing \$0.165/hr/high-cpu instance [13]) and cache instances (costing \$0.45/hr/high-memory instance [13]). Suppose load drops by a factor of 4. If we only scale the application tier (by a factor of 4), then we can save 45% of the peak operational cost. But if we also scale the caching tier (assuming a modest 50% cache reduction based on Figure 6.4), then we can save 65% of the peak operational cost. This is an additional 44% savings relative to only scaling the application tier.

6.3.4 Experimental results

In order to validate our theoretical results, we experimentally determine the lowest memcached size that we can afford without violating our SLA. We do this by monitoring the mean response time, T_{avg}, for different memcached sizes, and then picking

the smallest memcached size which keeps T_{avg} below T_{SLA}. The crosses in Figure 6.4 show our experimental results, which agree with the theoretical results. We also investigated 99th percentile response times and preliminary results show similar cache savings as in Figure 6.4.

6.4 CacheScale

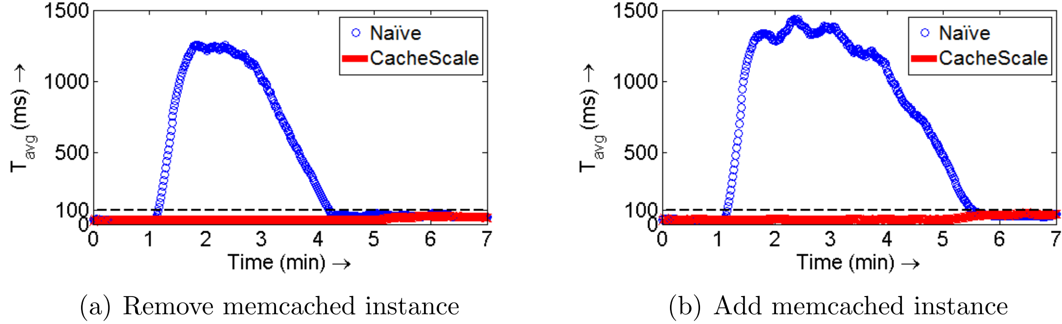
Thus far, we have investigated how small the cache can be, assuming it contains the most popular data items. However, it is not obvious how to retain the most popular data items when scaling the caching tier, because the cached objects are distributed over the cache instances for load balancing. When the number of cache instances is reduced, popular items may be lost. We use consistent hashing in the libMemcached library [1] to ensure that only the data in the newly added/removed cache instance needs to change.

6.4.1 Scaling down

If we naïvely remove a cache instance, we immediately lose all the cached data that was stored on the instance. This causes a drop in hit rate, which increases the request rate to the database as well as the average response times. For example, consider the case in Figure 6.5(a) where we scale down from 4 to 3 cache instances at the 1 minute mark. In this experiment, we choose $\alpha = 1.2$ to limit the amount of cached data that is lost. We also pessimistically force cache misses to result in database disk accesses by avoiding the database page cache. As observed in Figure 6.5(a), the Naïve solution (the circles) creates a spike in average response time.

To avoid this spike in response times, we propose *gradually* migrating the popular data off of the instance to be removed. We refer to this approach as *CacheScale*. Conceptually, *CacheScale* treats the instance to be removed as a second level cache for some period of time. If we miss in the rest of the cache, then *CacheScale* queries this instance before going to the database. This will naturally migrate the popular items from this "retiring" instance to the rest of the caching tier.

This leads to the question of how long to keep an instance in this retiring state. Intuitively, we want to stay in this retiring state until the probability of querying this instance is low enough that we can achieve our target hit rate, as calculated from Equation (6.3), even after this instance is removed. For each item i residing on this instance, let p_i denote the probability of requesting that item. Suppose we



(a) Remove memcached instance

(b) Add memcached instance

Figure 6.5: *CacheScale* is significantly better than Naïve for (a) scaling down and (b) scaling up the caching tier. Here, the dashed line indicates $T_{SLA} = 100ms$.

have received N requests since entering this retiring state. Then the probability that item i has not yet migrated to the rest of the caching tier is $(1 - p_i)^N$. Thus, the probability that the $(N + 1)^{th}$ request queries this instance is:

$$\sum_{\text{item } i \in \text{instance}} p_i(1 - p_i)^N. \tag{6.5}$$

Assuming that all data items are equally distributed among the cache instances, we can reasonably predict how large N needs to be so that the probability of querying this instance is low enough to achieve our target hit rate.

In Figure 6.5(a), we see that *CacheScale* (solid line) eliminates the spike in response times. Here, the retiring instance acts as a "second-level cache" between the 1 minute and 5 minute marks, approximated by Equation (6.5), at which point it is entirely removed.

6.4.2 Scaling up

If we naïvely add a cache instance, it has a "cold" cache and all requests to that instance result in misses. For example, consider the case in Figure 6.5(b) where we scale up from 3 to 4 cache instances at the 1 minute mark. We see that the Naïve solution (the circles) creates a spike in average response time.

To avoid this spike in response times, *CacheScale* gradually migrates the most popular data from the rest of the caching tier to this new instance. That is, when we miss in this new instance, *CacheScale* queries the rest of the caching tier. This will naturally warm up the "hot" data without needing to go to the database. In Figure 6.5(b), we see that *CacheScale* (solid line) eliminates the spike in response

times. Here, the new instance warms up between the 1 minute and 5 minute marks, approximated by Equation (6.5), using the rest of the caching tier.

Importantly, using *CacheScale* for both adding and removing a cache instance keeps our response times below the 100ms SLA. We also find the same behavior when looking at 99th percentile response times. The 99th percentile under the Naïve solution increases to about 16 seconds, but stays below 700ms under *CacheScale*. The above findings warrant a more comprehensive evaluation of *CacheScale*; we defer this to future work.

6.4.3 Alternative approaches

While *CacheScale* is a very simple policy, additional benefits may be realized through more dramatic re-architecting of the caching tier. For example, by exposing the least-recently-used list of cached keys at each memcached instance, more intelligent scaling of cache instances may be possible. Further, integration of non-volatile storage, such as flash, may allow removing a cache instance without requiring the migration of hot data from other instances on reactivation. However, this approach will require additional functionality to determine the consistency of cached data upon reactivation.

6.5 Chapter Summary

In this chapter we consider dynamic provisioning of the caching tier. While dynamic provisioning of the stateless application tier has been proposed in many research papers, there has been almost no discussion of scaling the stateful caching tier. The "seemingly small" benefits associated with scaling the caching tier coupled with the fear of severe performance degradation due to cache misses has deterred this research.

We demonstrate that, given the skewed popularity distribution for data accesses, significant savings can be obtained by scaling the caching tier under long-term load variations (see Section 6.3). In fact, a 50% drop in load can lead to 90% savings in the cache size. In order to realize these savings, we present *CacheScale* (see Section 6.4), a simple cache management policy that avoids the performance problems associated with scaling the stateful caching tier. *CacheScale* works by proactively moving the "hot" cached data items before scaling the caching tier. Our preliminary experiments with *CacheScale* demonstrate that we can successfully scale the caching tier without violating response time SLAs. By combining *CacheScale* with stateless scaling solutions, such as *AutoScale*, we can realize fully load-proportional systems.

Chapter 7

Recursive Renewal Reward

In this chapter we consider the M/M/k/setup system, which is very useful for modeling and analyzing multi-server systems with setup times. The M/M/k/setup system, where there is a penalty for turning servers on, is common in data centers, call centers and manufacturing systems. While the M/M/1/setup was exactly analyzed in 1964, no exact analysis exists to date for the M/M/k/setup with $k > 1$.

We present the first exact, closed-form analysis for the M/M/k/setup. Our analysis is made possible by our development of a new technique, Recursive Renewal Reward (RRR), for solving Markov chains with a repeating structure. RRR uses ideas from renewal reward theory and busy period analysis to obtain closed-form expressions for metrics of interest such as the transform of time in system and the transform of power consumed by the system. The simplicity, intuitiveness, and versatility of RRR makes it useful for analyzing Markov chains far beyond the M/M/k/setup. In general, RRR should be able to reduce the analysis of any 2-dimensional Markov chain which is infinite in at most one dimension and repeating to the problem of solving a system of polynomial equations.

We introduce the M/M/k/setup system in Section 7.1, and discuss prior work on the analysis of M/M/k/setup in Section 7.2. We formalize the M/M/k/setup model in Section 7.3. We then present the RRR technique in Section 7.4. We illustrate the use of RRR by analyzing the M/M/1/setup in Section 7.5. We then discuss how RRR can be used to analyze the M/M/k/setup in Section 7.6. Finally, we discuss how RRR can be used to analyze Markov chains beyond the M/M/k/setup in Section 7.7. We conclude with a summary of this chapter in Section 7.8.

7.1 Introduction

Setup times (a.k.a. exceptional first service) are a fundamental component of computer systems and manufacturing systems, and therefore they have always played an important role in queueing theoretic analysis. In manufacturing systems it is very common for a job that finds a server idle to wait for the server to "warm up" before service is initiated. In retail and hospitals, the arrival of customers may necessitate bringing in an additional human server, which requires a setup time for the server to arrive. In computer systems, setup times are once again at the forefront of research, as they are the key issue in dynamic capacity provisioning for data centers.

In data centers, it is desirable to turn idle servers off, or reallocate the servers, to save power. This is because idle servers burn power at 60–70% of the peak rate, so leaving servers on and idle is wasteful [26]. Unfortunately, most companies are hesitant to turn off idle servers because the setup time needed to restart these servers is very costly; the typical setup time for servers is 200 seconds, while a job's service requirement is typically less than 1 second [107, 48]. Not only is the setup time prohibitive, but power is also burned at peak rate during the entire setup period, although the server is still not functional. Thus it is not at all obvious that turning off idle servers is advantageous.

Surprisingly, for all the importance of setup times, very little is known about their analysis. The M/G/1 with setup times was analyzed in 1964 by Welch [194]. The analysis of an M/M/k system with setup times, which we refer to as *M/M/k/setup*, however, has remained elusive, owing largely to the complexity of the underlying Markov chain (Figure 7.1 shows an M/M/k/setup with exponentially distributed setup times). In 2010, various analytical approximations for the M/M/k/setup were proposed in [68]. These approximations work well provided that either load is low or the setup time is low. The M/M/∞/setup was also analyzed in [68] and found to exhibit product form. Other than the above, no progress has been made on the M/M/k/setup. Section 7.2 describes related prior work, including existing methods for solving general Markov chains with a repeating structure.

This work is the first to derive an exact, closed-form solution for the M/M/k/setup. We obtain the Laplace transform of response time, the z-transform of power consumption, and other important metrics for the M/M/k/setup.

Our solution is made possible by our development of a new technique for solving Markov chains with a repeating structure – Recursive Renewal Reward (RRR). RRR is based on using renewal reward theory [160, Chapter 7] to obtain the metrics of interest, while utilizing certain recursion theorems about the chain. Unlike matrix-

analytic methods [111], RRR does not require finding the "rate" matrix. Another feature of RRR is that it is simple enough to be taught in an elementary stochastic processes course.

In general, RRR should be able to reduce the analysis of any 2-dimensional Markov chain which is finite in one dimension, say the vertical dimension, and infinite (with repeating structure) in the other (horizontal dimension) to the problem of solving a system of polynomial equations. Further, if in the repeating portion all horizontal transitions are skip-free and all vertical transitions are unidirectional, the resulting system of equations will be at most quadratic, yielding a closed-form solution (see Section 7.7 and Figure 7.3 for more details). We thus anticipate that RRR will prove useful to other researchers in analyzing many new problems.

7.2 Prior Work

The few papers that have looked at the M/M/k/setup are discussed in Section 7.1. We now discuss papers that have considered repeating Markov chains and have proposed techniques for solving these. We then comment on how these techniques might or might not apply to the M/M/k/setup.

7.2.1 Matrix-analytic based approaches

Matrix-analytic methods are a common approach for analyzing Markov chains with repeating structure. Such approaches are typically numerical, generally involving iteration to find the rate matrix, R. These approaches do not, in general, lead to closed forms or to any intuition, but are very useful for evaluating chains under different parameters.

There are cases where it is known that the R matrix can be stated explicitly [111]. This typically involves using a combinatorial interpretation for the R matrix. As described in [111], the class of chains for which the combinatorial view is tractable is narrow. However, in [178], the authors show that the combinatorial interpretation extends to a broader class of chains. Their class does not include the M/M/k/setup, however, which is more complicated because the transition (setup) rates are not independent of the number of jobs in system. In [177], the authors derive the explicit rate matrix for a broader class of chains than that in [178], in terms of infinite sums. The authors also provide an algorithm for deriving the rate matrix in closed form for a specific subclass of the chains that they consider. Though not shown in their

paper, we believe that the algorithm proposed by the authors provides a closed-form rate matrix for the M/M/k/setup.

7.2.2 Generating function based approaches

Generating functions have also been applied to solve chains with a repeating structure. Like matrix-analytic methods these are not intuitive: Generating function approaches involve guessing the form of the solution and then solving for the coefficients of the guess, often leading to long computations. In theory, they can be used to solve very general chains (see, for example, [4]). We initially tried applying a generating function approach to the M/M/2/setup and found it to be incredibly complex and without intuition. This led us to seek a simpler and more intuitive approach.

7.2.3 M/M/k with vacations

Many papers have been written about the M/M/k system with vacations (see, for example, [199, 197, 171, 116]). While the Markov chain for the M/M/k with vacations looks similar to the M/M/k/setup, the dynamics of the two systems are very different. A server takes a vacation as soon as it is idle and there are no jobs in the queue. By contrast, a setup time is initiated by jobs arriving to the queue. In almost all of the papers involving vacations, the vacation model is severely restricted, allowing only a fixed group of servers to go on vacation at once. This is very different from our system in which any number of servers may be in setup at any time. The model in [116] comes closest to our model, although the authors use generating functions and assume that *all* idle servers are on vacation, rather than one server being in setup for each job in queue, which makes the transitions in their chain independent of the number of jobs.

7.2.4 Restricted models of M/M/k with setup

There have been a few papers [5, 23, 68] that consider a very restricted version of the M/M/k/setup, wherein at most one server can be in setup at a time. There has also been prior work [136] that considers an M/M/k system wherein a fixed subset of servers can be turned on and off based on load. The underlying Markov chains for all of these restricted systems are analytically tractable and lead to very simple

closed-form expressions, since the rate at which servers turn on is always fixed. Our M/M/k/setup system is more general, allowing any number of servers to be in setup. This makes our problem much more challenging.

7.2.5 How our work differs from all of the above

To the best of our knowledge, we are the first to derive exact closed-form results for the M/M/k/setup problem, with $k > 1$. Our solution was made possible by our new RRR technique. RRR results in exact solutions, does not require any iteration, and does not involve infinite sums. Importantly, RRR is highly intuitive and very easy to apply.

7.3 Model

In our model jobs arrive according to a Poisson process with rate λ and are served at rate $\mu = \frac{1}{E[S]}$, where S denotes the job size and is exponentially distributed. For stability, we assume that $k \cdot \mu > \lambda$, where k is the number of servers in the system.

In the M/M/k/setup system, each of the k servers is in one of three states: *off*, *on* (being used to serve a job), or *setup*. When a server is on or in setup, it consumes peak power of P_{peak} watts. When a server is off, it consumes zero power. Thus, when servers are not in use, they are immediately turned off to save power. Every arriving job that comes into the system picks an off server, if one exists, and puts it into setup mode; the job then joins the queue. We use I to denote the setup times, with $E[I] = \frac{1}{\alpha}$. Unless stated otherwise, we assume that setup times are exponentially distributed. When a job completes service at a server, say server s_1, and there are no remaining jobs left in the queue, then server s_1 is immediately turned off. However, if the queue is not empty, then server s_1 is not turned off, and the job at the head of the queue is directed to server s_1. Note that if the job at the head of the queue was already waiting on another server, say server s_2, in setup mode, the job at the head of the queue is still directed to server s_1. At this point, if there is a job in the queue that did not setup an off server on arrival (because there were no off servers), then server s_2 continues to be in setup for this job. If no such job exists in the queue, then server s_2 is turned off.

The Markov chain for the M/M/k/setup system is shown in Figure 7.1. Each state is denoted by the pair (i, j), where i is the number of on servers, and j is the

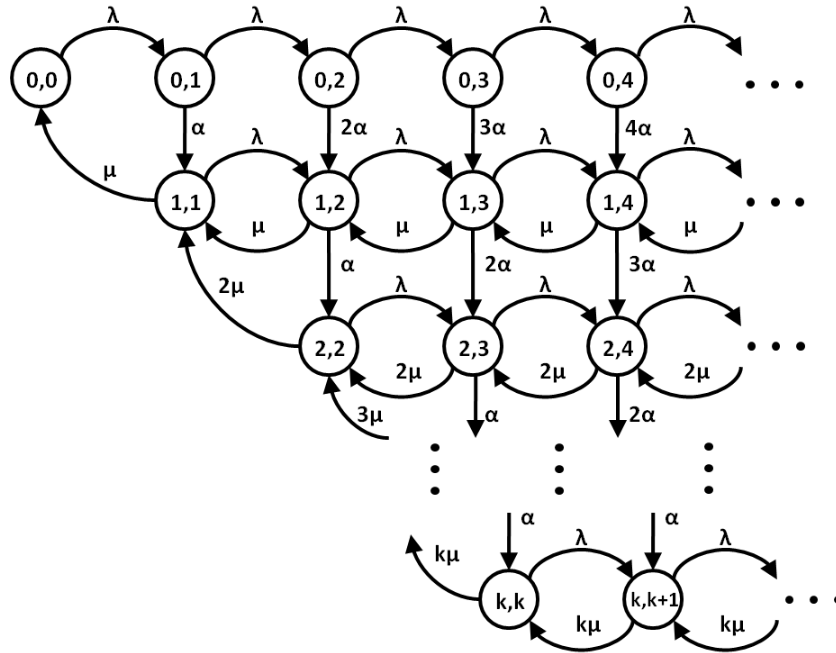


Figure 7.1: M/M/k/setup Markov chain. Each state is denoted by the pair (i, j), where i is the number of on servers, and j is the number of jobs in the system. The number of servers in setup is $\min\{j - i, k - i\}$.

number of jobs in the system. Thus, the number of servers in setup is $\min\{j-i, k-i\}$. Note that the Markov chain is infinite in one dimension.

7.4 The Recursive Renewal Reward technique

In this section we provide a high-level description of our new Recursive Renewal Reward (RRR) technique, which yields exact, closed-form solutions for a range of Markov chains, including the M/M/k/setup. In the next section, we illustrate the use of RRR by analyzing the M/M/1/setup. We also analyze the M/M/2/setup and the M/M/3/setup via RRR, but we will not discuss them here; we refer the reader to our papers [62, 63] for the full analysis.

The RRR technique works by deriving the *expected "reward" earned per unit time in a Markov chain*, where the reward could be any quantity of interest. In the context of our M/M/k/setup problem, the reward earned at time t, $R(t)$, could be the number of jobs in system at time t, the square of the number of jobs in system, the current power usage, the number of servers that are on, or any other reward that

can be expressed as a function of the state of the Markov chain.

To analyze the average rate of earning reward, we designate a *renewal state*, say $(0,0)$,[1] which we call the *home state*, and then consider a *renewal cycle* to be the process of moving from the home state back to the home state. By renewal reward theory [160, Chapter 7], the average rate of earning reward is the same as the mean reward earned over a renewal cycle, which we denote by $\mathcal{R}$, divided by the mean length of the renewal cycle, denoted by $\mathcal{T}$.

$$\text{Average rate of earning} = \frac{\mathcal{R}}{\mathcal{T}} = \frac{E\left[\int_{\text{cycle}} R(t)dt\right]}{E\left[\int_{\text{cycle}} 1dt\right]}$$

For example, if the goal is to find the mean number of jobs, $E[N]$, for our chain, we simply define $R(t)$ to be the number of jobs at time t, which can be obtained from the state of the Markov chain at time t.

It turns out that the quantities $\mathcal{T}$ and $\mathcal{R}$ are very easy to compute! Consider a Markov chain, such as that in Figure 7.2. The *repeating portion* of the chain is shown in gray. There are a finite number of *border states* which sit at the edge of the repeating chain and are colored black. We will see that computing $\mathcal{T}$ and $\mathcal{R}$ basically reduces to writing one equation for each border state[2]. For the case of $\mathcal{T}$, we will need the mean time to move one step left from each border state. For the case of $\mathcal{R}$, we will need the mean reward earned when moving one step left from each border state. Computing these border state quantities is made very easy via some neat recursion theorems. We demonstrate this process in the M/M/1/setup example below. There are a few details which we will defer until after the example. For instance, in general, it is necessary to also add equations for the non-repeating portion of the Markov chain. See Sections 7.6 and 7.7 for more details on the RRR technique.

7.5 M/M/1/setup

In this section we illustrate the RRR technique by applying it to the M/M/1/setup system, whose Markov chain is shown in Figure 7.2. Here, the state of the system is

[1] In principle any state can be chosen as the renewal state, but some states allow for an easier (or shorter) analysis.

[2] Several techniques in the literature such as matrix-analytic methods [111] and stochastic complementation [159] also deal with border states, although none of them involve renewal-reward theory.

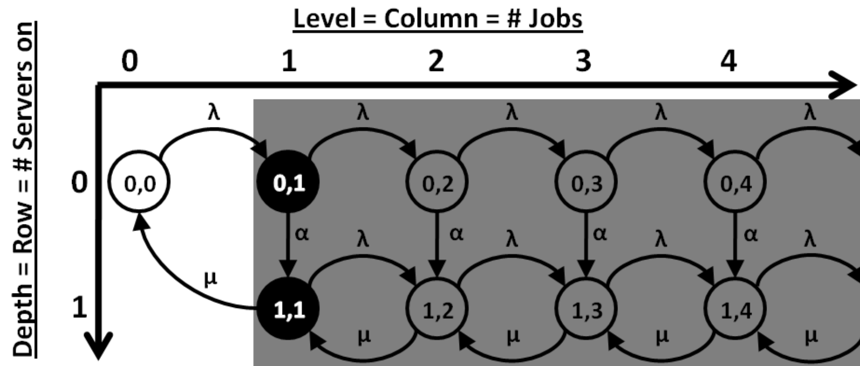


Figure 7.2: M/M/1/setup Markov chain with the repeating portion highlighted in gray and the border states shaded black.

represented as (i, j), where $i \in \{0, 1\}$ is the number of servers on and $0 \leq j < \infty$ is the number of jobs in the system. In general, i represents the *depth* (or row number) of the state, and j represents the *level* (or column number) of the state. We start by deriving $E[N]$, the mean number of jobs, and then move to more complex metrics. We choose the renewal state to be $(0, 0)$ and we define the reward earned at time t, $R(t)$, to be $N(t)$, the number of jobs in the system at time t. As explained in Section 7.4, all we need is $\mathcal{T}$ and $\mathcal{R}$.

7.5.1 Deriving $\mathcal{T}$ via $\mathbf{T^L_{0,1}}$ and $\mathbf{T^L_{1,1}}$

$\mathcal{T}$ is the mean time to get from our home state $(0, 0)$ back to $(0, 0)$. This can be viewed as $\frac{1}{\lambda}$, the mean time until we leave $(0, 0)$ (which takes us to $(0, 1)$) plus the mean time to get home from $(0, 1)$. We make the further observation that the mean time to get home from $(0, 1)$ is equal to $T^L_{0,1}$ (using notation from Table 7.1), the mean time to move left one level from $(0, 1)$ (since moving left can only put us in $(0, 0)$). We thus have:

$$\mathcal{T} = \frac{1}{l} + T^L_{0,1} \tag{7.1}$$

We now need an equation for $T^L_{0,1}$ for the border state $(0, 1)$, which will require looking at the other border state, $(1, 1)$, as well. Starting with border state $(1, 1)$, it is clear that $T^L_{1,1}$ is simply the mean length of an M/M/1 busy period, B_1. Thus:

$$T^L_{1,1} = B_1 = \frac{1}{\mu - l} \tag{7.2}$$

$T^L_{0,1}$ involves waiting in state $(0,1)$ for expected time $\frac{1}{\alpha+\lambda}$, before conditioning on where we transition to next. If we go to state $(1,1)$ we need an additional $T^L_{1,1}$. However if we go to state $(0,2)$ we need to add on the time to move one step left from $(0,2)$ (which by Figure 7.2 takes us to $(1,1)$) and then an additional $T^L_{1,1}$:

$$T^L_{0,1} = \frac{1}{l+\alpha} + \frac{\alpha}{l+\alpha} \cdot T^L_{1,1} + \frac{l}{l+\alpha}\left(T^L_{0,2} + T^L_{1,1}\right) \tag{7.3}$$

It is now time to invoke one of our recursion theorems:

Theorem 7.1 (Recursion theorem for mean time)**.**
For the M/M/k/setup, the mean time to move one step left from state (i,j), $T^L_{i,j}$, *is the same for all* $j \geq k$.

Proof. For any $j \geq k$, observe that when moving one step left from any state (i,j), we only visit states with level j or greater, until the final transition to level $j-1$. Hence, $T^L_{i,j}$ depends only on the structure of the "subchain" of the M/M/k/setup consisting of levels $\{j, j+1, \ldots\}$, including transition rates to level $j-1$. Now consider the subchain for each $j \geq k$; these subchains are isomorphic, by the fact that the chain is repeating from level k onward. Hence, the time to move one step left is the same regardless of the initial level $j \geq k$. □

Using Theorem 7.1, we replace $T^L_{0,2}$ in Equation (7.3) with $T^L_{0,1}$ to get:

$$T^L_{0,1} = \frac{1}{l+\alpha} + \frac{\alpha}{l+\alpha} \cdot T^L_{1,1} + \frac{l}{l+\alpha}\left(T^L_{0,1} + T^L_{1,1}\right) \tag{7.4}$$

Finally, noting that $T^L_{1,1} = B_1$ from Equation (7.2), we have that:

$$T^L_{0,1} = \frac{1}{l+\alpha} + \frac{\alpha}{l+\alpha} \cdot B_1 + \frac{l}{l+\alpha}\left(T^L_{0,1} + B_1\right)$$

$$\implies T^L_{0j} = T^L_{0,1} = \frac{1 + (l+\alpha)B_1}{\alpha} \tag{7.5}$$

Substituting $T^L_{0,1}$ from above into Equation (7.1) gives us $\mathcal{T}$:

$$\mathcal{T} = \frac{\mu(l+\alpha)}{l\alpha(\mu - l)} \tag{7.6}$$

Variable	Description
$\mathcal{T}$	Mean length of the renewal cycle
$\mathcal{R}$	Mean reward earned during a renewal cycle
$T^L_{i,j}$	Mean time until we first move one level left of (i,j), starting from (i,j)
$R^L_{i,j}$	Mean reward earned until we first move one level left of (i,j), starting from (i,j)
$p^L_{i\to d}$	Probability that after we first move one level left from state (i,j), we are at depth d
B_k	Mean length of an $M/M/k$ busy period

Table 7.1: Variables used in our analysis of $E[N]$.

7.5.2 Deriving $\mathcal{R}$ via $\mathbf{R^L_{0,1}}$ and $\mathbf{R^L_{1,1}}$

$\mathcal{R}$ denotes the reward earned in moving from $(0,0)$ back to $(0,0)$. Observing that we earn 0 reward in state $(0,0)$ (because there are no jobs in the system in that state), and observing that from state $(0,0)$ we can only next move to $(0,1)$, we have (using notation from Table 7.1):

$$\mathcal{R} \;=\; R^L_{0,1} \tag{7.7}$$

It now remains to compute the reward earned in moving one step left from $(0,1)$, which will require looking at the other border state, $(1,1)$, as well.

To do this, we invoke another recursion theorem, which again holds for any M/M/k/setup system:

Theorem 7.2 (Recursion theorem for mean reward)**.**
For the M/M/k/setup, the mean reward earned in moving one step left from state $(i,j+1)$, $R^L_{i,j+1}$, satisfies $R^L_{i,j+1} = R^L_{i,j} + T^L_{i,j}$ for all $j \geq k$, where the reward tracks the number of jobs in the system.

Proof. Consider the process of moving one step left from a given state (i,j) where $j \geq k$. At the same time, consider the same process where everything is shifted over one level to the right, so that the initial state is $(i,j+1)$ At any point in time, the number of jobs seen by the second process is exactly one greater than that seen by the first process. Therefore, the total number of jobs accumulated (total reward) during the second process is $T^L_{i,j}$ greater than that of the first process, since the duration of both processes is $T^L_{i,j}$ by Theorem 7.1. □

Applying Theorem 7.2 to the Markov chain shown in Figure 7.2, we have:

$$R^L_{1,1} = \frac{1}{l+\mu} \cdot 1 + \frac{\mu}{l+\mu} \cdot 0 + \frac{l}{l+\mu} \left(R^L_{1,2} + R^L_{1,1}\right) \tag{7.8}$$

$$= \frac{1}{l+\mu} + \frac{l}{l+\mu} \left((R^L_{1,1} + T^L_{1,1}) + R^L_{1,1}\right)$$

$$= \frac{1}{l+\mu} + \frac{l}{l+\mu} \left((R^L_{1,1} + B_1) + R^L_{1,1}\right) \quad \text{(from Equation (7.2))}$$

$$\implies R^L_{1,1} = \frac{1 + lB_1}{\mu - l} \tag{7.9}$$

Similarly, for border state $(0,1)$, we have:

$$R^L_{0,1} = \frac{1}{l+\alpha} \cdot 1 + \frac{\alpha}{l+\alpha} \cdot R^L_{1,1} + \frac{l}{l+\alpha} \left(R^L_{0,2} + R^L_{1,1}\right)$$

$$= \frac{1}{l+\alpha} + \frac{\alpha}{l+\alpha} \cdot R^L_{1,1} + \frac{l}{l+\alpha} \left((R^L_{0,1} + T^L_{0,1}) + R^L_{1,1}\right) \quad \text{(from Theorem 7.2)}$$

$$\implies R^L_{0,1} = \frac{1 + lT^L_{0,1} + (l+\alpha)R^L_{1,1}}{\alpha}. \tag{7.10}$$

Substituting $R^L_{0,1}$ from above into Equation (7.7) gives us $\mathcal{R}$:

$$\mathcal{R} = \frac{\mu(l+\alpha)(\mu - l + \alpha)}{\alpha^2(\mu - l)^2} \tag{7.11}$$

7.5.3 Deriving E[N]

Since $E[N] = \frac{\mathcal{R}}{\mathcal{T}}$, combining Equation (7.6) and Equation (7.11), we get:

$$E[N] = \frac{\mathcal{R}}{\mathcal{T}} = \frac{l}{\alpha} + \frac{l}{\mu - l} \tag{7.12}$$

Variable	Description
$\dot{\mathcal{R}}$	Mean reward earned (for z-transform) during a renewal cycle
$\dot{\mathcal{E}}$	Mean reward earned (for transform of power) during a renewal cycle
$\dot{R}^L_{i,j}$	Mean reward earned (for z-transform) until we first move one level left of (i,j), starting from (i,j)
$\dot{E}^L_{i,j}$	Mean reward earned (for z-transform of power) until we first move one level left of (i,j), starting from (i,j)

Table 7.2: Variables used in our transform analyses.

The second term in the right hand side of Equation (7.12) can be identified [103] as the mean number of jobs in an M/M/1 system (without setup). Thus, Equation (7.12) is consistent with the known decomposition property for the M/M/1/setup system [194].

7.5.4 Deriving $\widehat{\mathbf{N}}(\mathbf{z})$ and $\widetilde{\mathbf{T}}(\mathbf{s})$

Deriving the z-transform of the number of jobs, $\widehat{N}(z) = E[z^N]$, is just as easy as deriving $E[N]$. The only difference is that our reward function is now $R(t) = z^{N(t)}$, where $N(t)$ is again the number of jobs in the system at time t. Thus

$$\widehat{N}(z) = E[z^N] = \frac{\dot{\mathcal{R}}}{\mathcal{T}},$$

where $\dot{\mathcal{R}} = E\left[\int_{\text{cycle}} z^{N(t)}dt\right]$ and $\mathcal{T}$ is the same as before.

We will again invoke a recursion theorem which applies to any M/M/k/setup (using notation from Table 7.2):

Theorem 7.3 (Recursion theorem for transform of reward)**.** *For the M/M/k/setup, $\dot{R}^L_{i,j+1} = z \cdot \dot{R}^L_{i,j}$, for all $j \geq k$, where $\dot{R}$ tracks the z-transform of the number of jobs in the system.*

Proof. The proof is identical to that of Theorem 7.2, except that in any moment in time the second process (starting in level $(i, j+1)$) earns z times as much reward as the first process (starting at (i,j)). □

Let us now express $\dot{\mathcal{R}}$ by conditioning on the first step from $(0,0)$:

$$\dot{\mathcal{R}} = \frac{1}{l} + \dot{R}^L_{0,1} \tag{7.13}$$

We again need one equation per border state:

$$\dot{R}^L_{1,1} = \frac{1}{l+\mu} \cdot z + \frac{l}{l+\mu}\left(z \cdot \dot{R}^L_{1,1} + \dot{R}^L_{1,1}\right)$$

$$\dot{R}^L_{0,1} = \frac{1}{l+\alpha} \cdot z + \frac{\alpha}{l+\alpha} \cdot \dot{R}^L_{1,1} + \frac{l}{l+\alpha}\left(z \cdot \dot{R}^L_{0,1} + \dot{R}^L_{1,1}\right)$$

Solving the above system and substituting $\dot{R}^L_{0,1}$ into Equation (7.13) allows us to express $\dot{\mathcal{R}}$ in closed form. This gives us $\widehat{N}(z)$, after some algebra, as follows:

$$\widehat{N}(z) = E[z^N] = \frac{\dot{\mathcal{R}}}{\mathcal{T}} = \frac{\alpha(\mu - l)}{(\mu - lz)(\alpha + l - lz)} \tag{7.14}$$

To get the Laplace transform of response time, $\widetilde{T}(s)$, we use the distributional Little's Law [98] (since M/M/1/setup is a First-In-First-Out system):

$$\widetilde{T}(s) = \widehat{N}\left(1 - \frac{s}{l}\right) = \frac{\alpha(\mu - l)}{(s+\alpha)(\mu + s - l)} \tag{7.15}$$

7.5.5 Deriving $\widehat{\mathbf{P}}(\mathbf{z})$

We now derive $\widehat{P}(z)$, the z-transform of the power consumed for the M/M/1/setup. The server consumes zero power when it is off, but consumes peak power, P_{peak} watts, when it is on or in setup. This time, the reward is simply the transform of the energy consumed over the renewal cycle, $\dot{\mathcal{E}} = E\left[\int_{\text{cycle}} z^{P(t)} dt\right]$, where $P(t)$ is the power consumed at time t. We begin with the recursive theorem for $\dot{E}^L_{i,j}$, just like we had Theorem 7.3 for $\dot{R}^L_{i,j}$.

Theorem 7.4 (Recursion theorem for transform of power)**.** *For the M/M/k/setup,* $\dot{E}^L_{i,j+1} = \dot{E}^L_{i,j} = T^L_{i,j} \cdot z^{k \cdot P_{peak}}$*, for all* $j \geq k$*.*

Proof. When $j \geq k$, all k servers are either on or in setup, putting power consumption at $k \cdot P_{peak}$. So the transform of power usage is $z^{k \cdot P_{peak}}$, yielding $\dot{E}^L_{i,j} = T^L_{i,j} \cdot z^{k \cdot P_{peak}}$. It then follows immediately from Theorem 7.1 that $\dot{E}^L_{i,j+1} = \dot{E}^L_{i,j}$. □

Theorem 7.4 gives us $\dot{E}^L_{i,j}$ in closed form, in terms of $T^L_{i,j}$. Following the usual renewal-reward approach, we get:

$$\widehat{P}(z) = E[z^P] = \frac{\dot{\mathcal{E}}}{\mathcal{T}} = \frac{\alpha(\mu - l) + l(\mu + \alpha) z^{P_{peak}}}{\mu(l + \alpha)} \tag{7.16}$$

7.6 M/M/k/setup

The M/M/k/setup chain shown in Figure 7.1 is analyzed similarly to M/M/1/setup. The only complication is that when moving one level left from a given state, the resulting row is non-deterministic. For example, when moving left from $(1, 3)$ in Fig. 7.1, we may end up in row 1 at $(1, 2)$ or row 2 at $(2, 2)$. We use $p^L_{i\to d}$ to denote the probability that once we move one level left from (i, j), we will be at depth d.[2] The following theorem proves that $p^L_{i\to d}$ is independent of j for all states (i, j) in the repeating portion.

Theorem 7.5 (Recursion theorem for probability)**.**
For the M/M/k/setup, for each $0 \le d \le k$ *and for each* $0 \le i \le k$, $p^L_{i\to d}$ *is the same for all* $j \ge k$.

Proof. Recall that $p^L_{i\to d}$ is the probability that, given that we start at depth i, we end at depth d when moving one step to the left, except when $j = k$ and $d \in \{k - 1, k\}$; in these cases we interpret $p^L_{i\to k}$ (or $p^L_{i\to k-1}$) as the probabilities that we first moved one step left by *transitioning out of a state* in depth k (or $k - 1$).

As with $T^L_{i,j}$, $p^L_{i\to d}$ depends only on the structure of the "subchain" consisting of levels $\{j, j + 1, \ldots\}$, including transition rates to level $j - 1$. Since for all $j \ge k$ the resulting subchains are isomorphic, $p^L_{i\to d}$ must be the same for all $j \ge k$. □

Thus, it suffices to compute $p^L_{i\to d}$ for the border states. For M/M/k/setup, the border states are (i, k), with $0 \le i \le k$.

In the M/M/k/setup, the non-repeating portion consists of $O(k^2)$ states. For $k = 1$, we did not have to explicitly write reward equations for the non-repeating states; these were implicitly folded into other equations (see, for example, the term

[2]The definition given for $p^L_{i\to d}$ applies in all cases except when $j = k$ and $d \in \{k - 1, k\}$. When $j = k$, we can never end in depth k when moving one step to the left; in this case, we interpret $p^L_{i\to k}$ (or $p^L_{i\to k-1}$) as the probability that we first moved one step left *by transitioning out of a state in depth* k (or $k - 1$).

in parentheses in Equation (7.13)). However, for arbitrarily large k, it is necessary to write reward equations for the states in the non-repeating portion. We use $R^H_{i,j}$ to denote the reward earned until we reach the home state, starting from state (i, j) in the non-repeating portion. The $R^H_{i,j}$ equations will be discussed in Section 7.6.3.

We illustrate the RRR technique for M/M/k/setup by deriving $\widehat{N_Q}(z)$, from which we can obtain $\widetilde{T}(s)$. For a detailed demonstration of this technique for the case of $k = 2$ and $k = 3$, see [62, 63]. One might think that analyzing the M/M/k/setup will require solving a kth degree equation. This turns out to be false. Analyzing the M/M/k/setup via RRR only requires solving equations which are, at worst, quadratic.

We choose $(k-1, k-1)$ to be the renewal state. Using RRR, $\dot{\mathcal{R}}$ can be expressed as:

$$\dot{\mathcal{R}} = \frac{1 + (k-1)\mu \dot{R}^H_{k-2,k-2} + \lambda \dot{R}^L_{k-1,k}}{l + (k-1)\mu} \tag{7.17}$$

We now derive the necessary $\mathbf{p^L_{i\to d}}$, $\dot{R}^L_{i,k}$, and $\dot{R}^H_{i,j}$ for computing $\dot{\mathcal{R}}$.

7.6.1 System of equations for $\mathbf{p^L_{i\to d}}$

The system of equations for $p^L_{i\to d}$ is as follows:[2]

$$p^L_{i\to i} = \frac{\lambda (p^L_{i\to i})^2 + i\mu}{\lambda + i\mu + (k-i)\alpha}, \quad (i < k) \tag{7.18}$$

$$p^L_{i\to d} = \frac{\lambda \left(\sum_{\ell=i}^{d} \left\{ (p^L_{i\to \ell})(p^L_{\ell\to d}) \right\} \right) + (k-i)\alpha (p^L_{i+1\to d})}{\lambda + i\mu + (k-i)\alpha}, \quad (i < d < k) \tag{7.19}$$

$$p^L_{i\to k} = 1 - \sum_{\ell=i}^{k-1} p^L_{i\to \ell}, \quad (i \le k) \tag{7.20}$$

The summation in Equations (7.19) above denotes the possible intermediate depths ℓ through which we can move from initial depth i to final depth d. The above system of equations involves linear and quadratic equations (including products of two unlike variables), and can be solved *symbolically* to find $p^L_{i\to d}$ in closed form (see [62] for more details).

7.6.2 Deriving $\dot{\mathbf{R}}^{\mathbf{L}}_{\mathbf{i,k}}$ for the repeating portion

The system of equations for $\dot{R}^L_{i,k}$ is as follows:

$$\dot{R}^L_{0,k} = \frac{z^k + \lambda\left(z\dot{R}^L_{0,k} + \sum_{\ell=1}^{k}\left\{(p^L_{0\to\ell})(\dot{R}^L_{\ell,k})\right\}\right) + k\alpha\dot{R}^L_{1,k}}{\lambda + k\alpha} \tag{7.21}$$

$$\dot{R}^L_{i,k} = \frac{z^{k-i} + \lambda\left(z\dot{R}^L_{i,k} + \sum_{\ell=i}^{k}\left\{(p^L_{i\to\ell})(\dot{R}^L_{\ell,k})\right\}\right) + (k-i)\alpha\dot{R}^L_{i+1,k}}{\lambda + i\mu + (k-i)\alpha}, \quad (0 < i < k) \tag{7.22}$$

$$\dot{R}^L_{k,k} = \frac{1 + \lambda(z\dot{R}^L_{k,k} + \dot{R}^L_{k,k})}{\lambda + k\mu} \tag{7.23}$$

In the above, we have used the fact that $\dot{R}^L_{i,k+1} = z\dot{R}^L_{i,k}$ from Theorem 7.3. The above system of linear equations can be easily solved to find $\dot{R}^L_{i,k}$ in closed form (see [62] for more details).

7.6.3 Deriving $\dot{\mathbf{R}}^{\mathbf{H}}_{\mathbf{i,j}}$ for the non-repeating portion

The system of equations for $\dot{R}^H_{i,j}$ is as follows:

$$\dot{R}^H_{0,j} = \frac{z^j + \lambda\dot{R}^H_{0,j+1} + j\alpha\dot{R}^H_{1,j}}{\lambda + j\alpha}, \quad (j < k-1) \tag{7.24}$$

$$\dot{R}^H_{i,j} = \frac{z^{j-i} + \lambda\dot{R}^H_{i,j+1} + i\mu\dot{R}^H_{i,j-1} + (j-i)\alpha\dot{R}^H_{i+1,j}}{\lambda + i\mu + (j-i)\alpha}, \quad (0 < i < j < k-1) \tag{7.25}$$

$$\dot{R}^H_{i,i} = \frac{1 + \lambda\dot{R}^H_{i,i+1} + i\mu\dot{R}^H_{i-1,i-1}}{\lambda + i\mu}, \quad (0 < i < k-1) \tag{7.26}$$

$$\dot{R}^H_{i,k-1} = \frac{z^{k-1-i} + \lambda\left(\dot{R}^L_{i,k} + \sum_{\ell=i}^{k}\left\{(p^L_{i\to\ell})(\dot{R}^H_{\ell,k-1})\right\}\right)}{\lambda + i\mu + (k-1-i)\alpha} + \frac{i\mu\dot{R}^H_{i,k-2} + (k-1-i)\alpha\dot{R}^H_{i+1,k-1}}{\lambda + i\mu + (k-1-i)\alpha}, \quad (i < k-1) \tag{7.27}$$

$$\dot{R}^H_{k-1,k-1} = 0 \tag{7.28}$$

Equations (7.24), (7.25), and (7.26), are simply based on the rate transitions in the non-repeating portion of the Markov chain. Equations (7.27) describe the rewards earned when starting in states in the non-repeating portion of the chain that can transition to the repeating portion of the chain via the border states. When we have an arrival in one of these states, we transition to the repeating portion of the chain, and after earning some reward, return to the non-repeating portion of the chain. Finally, Equation (7.28) guarantees that any transition to state $(k-1, k-1)$ will end the renewal cycle. The above system of linear equations can again be easily solved to find $\dot{R}^H_{i,j}$ in closed form (see [62] for more details).

After solving for $p^L_{i\to d}$, $\dot{R}^L_{i,k}$ and $\dot{R}^H_{i,j}$, we can derive $\dot{\mathcal{R}}$, and consequently $\widehat{N_Q}(z)$, via Equation (7.17). $\widetilde{T}(s)$ can then be derived by using the fact $\widetilde{T}(s) = \widetilde{T_Q}(s) \cdot \frac{\mu}{s+\mu} = \widehat{N_Q}\left(1 - \frac{s}{l}\right) \cdot \frac{\mu}{s+\mu}$.

We applied the above steps to obtain a closed-form expression for $\widehat{N_Q}(z)$ for the M/M/2/setup and the M/M/3/setup. We refer the reader to [62, 63] for full details.

7.7 The Generalized Recursive Renewal Reward technique

The RRR technique can be applied to a very broad class of Markov chains beyond just the M/M/k/setup. In general, RRR can reduce the analysis of any 2-dimensional, irreducible Markov chain which is repeating and infinite in one dimension (as shown in Figure 7.3(a)) to the problem of solving a system of polynomial equations. Further, if in the repeating portion all horizontal transitions are skip-free and all vertical transitions are unidirectional (as shown in Figure 7.3(b)), the resulting system of equations will be of degree at most two, yielding a closed-form solution. In this section we explain the application of the RRR technique to general repeating Markov

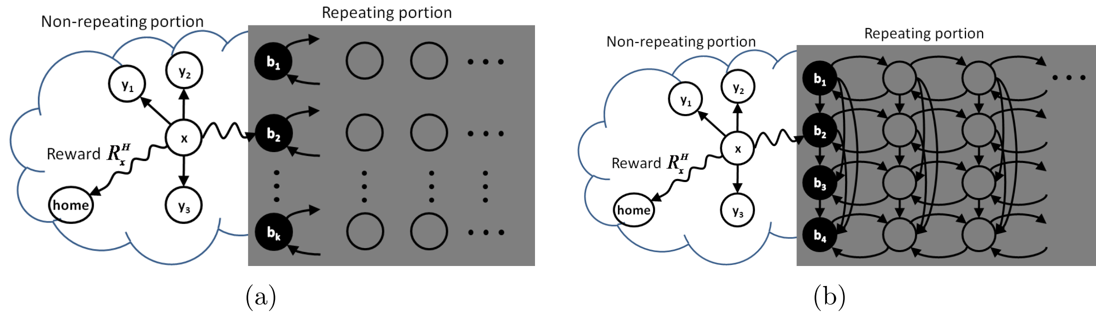


Figure 7.3: Figure 7.3(a) depicts the class of Markov chains that can be analyzed via RRR. The repeating portion is highlighted in gray and the border states, b_i, are shaded black. Note that y_i are the neighbors of x. Figure 7.3(b) depicts the more restrictive class of Markov chain that can be analyzed in closed-form via RRR. In this class, the horizontal transitions are skip-free and the vertical transitions are unidirectional.

chains and also provide justification for the above claims regarding Figures 7.3(a) and 7.3(b). Throughout we will assume that the reward earned at a state, (i, j), is an affine function of i and j.

In order to apply RRR, we first partition the Markov chain into a finite non-repeating portion and an infinite repeating portion as in Figure 7.3(a); in principle, this partition is not unique. Then, we fix a renewal point, or home state, within the non-repeating portion. For each state, x, in the non-repeating portion of the chain, we write an equation for the mean reward, R_x^H, earned in traveling from x to the home state. Each R_x^H is a sum of the mean reward during our residence in x and a weighted linear combination of the rewards R_y^H, where y is a neighbor of x, as in Figure 7.3(a). We refer to this finite set of linear equations for the R_x^Hs as (**Ia**). Since the chain is irreducible, at least one state in the non-repeating portion of the chain transitions directly to a state in the repeating portion of the chain. We refer to the states in the repeating portion that are directly accessible from the non-repeating portion as *border states*. These are shown as b_i in Figure 7.3(a). We next write a set of equations for the mean reward earned in traveling from each border state to the home state; call this set (**Ib**). Equation sets (**Ia**) and (**Ib**) together form the linear system of equations (**I**).

Within (**Ib**), the mean reward earned when returning home from each border state b consists of two parts: (i) the mean reward earned from the time we enter b until we leave the repeating portion, and (ii) the mean reward earned from when we first exit the repeating portion until returning home. Note that (ii) is simply a weighted linear

combination of R_x^Hs where the weights form the probability distribution over the set of states in the non-repeating portion that we transition to (same as the $p_{i\to d}^L$s). For (i), the reward equation can be expressed as a weighted linear combination of the rewards for the neighbors of b in the repeating portion. The fact that the chain has a repeating structure allows us to express the reward from any state in the repeating portion as a linear combination of the rewards of the border states by using "recursion theorems" (similar to Theorems 7.2 and 7.3). We also need the probability distribution over the set of states we transition to when we move left (the $p_{i\to d}^L$s). At this point, to write the equations in (**Ib**), we require solving the $p_{i\to d}^L$s. We refer to the system of equations for $p_{i\to d}^L$s as (**II**).

In writing the equations for $p_{i\to d}^L$s, we again use recursion theorems (similar to Theorem 7.5) that exploit the repeating structure of the Markov chain. However, this time, the equations need not be linear. This is because when moving left to depth d from depth i, we might transition through various intermediate depths. Thus, $p_{i\to d}^L$ will involve several other probability terms. Unlike rewards where we sum up intermediate terms, for probability we take a product of the intermediate terms, leading to a system of higher order polynomial equations, (**II**). Note that (**II**) does not depend on (**I**), and can be solved independently. Once we get the $p_{i\to d}^L$s by solving (**II**), we substitute these back (as constants) into the set of linear equations (**Ib**). The sets of linear equations (**Ib**) and (**Ia**) can now be jointly solved using standard techniques such as symbolic matrix inversion. This yields the mean reward earned during a renewal cycle from home to home; mean time is found analogously.

In the case of Markov chains as shown in Figure 7.3(b), the probability equations, (**II**), will be of degree at most two, as in Section 7.6.1. This is because skip-free horizontal transitions guarantee that the probability $p_{i\to d}^L$ can be expressed as a linear sum of products of only two intermediate terms of the form $p_{i\to \ell}^L \cdot p_{\ell\to d}^L$, where ℓ represents the intermediate depths that we can transition to in going from i to d (as in Section 7.6.1). Further, the unidirectional vertical transitions guarantee that $i \leq \ell \leq d$, which ensures that the intermediate probability terms do not lead to higher-order dependencies between each other. Thus, the probabilities can be derived in closed-form by solving quadratic equations (including products of two unlike terms) in a particular "bottom-up" order as explained in [62].

7.8 Chapter Summary

In this chapter we develop a new analysis technique, Recursive Renewal Reward (RRR), which allows us to solve the M/M/k/setup class of Markov chains. The M/M/k/setup allows us to model multi-server systems with setup times, and is thus very useful for understanding the effects of setup time. Our analysis technique, RRR, is very intuitive, easy to apply, and can be used to analyze many important Markov chains that have a repeating structure. RRR combines renewal reward theory with the development of recursion theorems for the Markov chain to yield exact, closed form results for metrics of interest such as the transform of time in system and the transform of power consumed by the system. RRR reduces the solution of the M/M/k/setup chains to solving k quadratic equations and a system of $O(k^2)$ linear equations. On an Intel Core i5-based processor machine we found RRR to be almost 5-10 times faster than the iterative matrix-analytic based methods, when using standard MATLAB implementations of both methods.

While we have only considered the M/M/k/setup in this chapter, we have also been able to use RRR for deriving exact, closed-form solutions for other important Markov chains with a repeating structure such as: (i) M/M/k/setup/delayed-off [65], wherein idle servers delay for a finite amount of time before turning off, (ii) M/M/k/setup/sleep, wherein idle servers can either be turned off or put to sleep, (iii) M/M/k/stag [68], wherein at most one server can be in setup, (iv) M/M/k/setup-threshold, wherein the servers are turned on and off based on some threshold for number of jobs in queue, (v) M/M/k/disasters, wherein the system can empty abruptly due to disasters, and (vi) M/E_2/k, where the job size distribution is Erlang-2. We have also been able to apply RRR to analyze other Markov chains such as: (i) M_t/M/1, where the arrival process is Poisson with a time dependent parameter, (ii) M/H_2/k, where the job size distribution is a 2-phase hyperexponential, and (iii) M^x/M/k, where there is a Poisson batch arrival process. In the above three cases, RRR reduces the analysis to solving a system of polynomial equations with degree > 2. In general, RRR should be able to reduce the analysis of any 2-dimensional Markov chain (with an affine reward function), which is finite in one dimension and infinite (with repeating structure) in the other, to solving a system of polynomial equations.

Chapter 8

Related Work

In this chapter we discuss related work in data center power management. In particular, we discuss the data center power management approaches of power-proportionality, energy-efficient server design, and consolidation and virtualization. These correspond to the approaches listed in Section 1.3. Prior work in dynamic server provisioning, which is the focus of this thesis, was discussed in detail in Section 2.2, and will not be reviewed here. However, we discuss how dynamic server provisioning compares with the other approaches.

8.1 Power-proportionality

Power-proportionality aims to reduce unnecessary power consumption in servers. A server is defined as being power-proportional if it consumes x% of its peak power when operating at a utilization of x%. Power-proportionality gained popularity in 2007 based on the research [26] by Barroso and Hölzle. The authors found that most servers spent the majority of their time operating in the 10% – 50% utilization range. Unfortunately, because of the high idle power of servers, they consumed much more than 10% – 50% of their peak power in this utilization range.

8.1.1 Lower server idle power

One of the approaches to power-proportionality is lowering the server idle power. Meisner et al. [131] show that reducing server idle power can lower server power consumption by up to 74%. However, reducing server idle power is a very challenging

task. While processor designers have been very successful at reducing the CPU idle power [109, 89], other server components still consume significant power when idle.

Reducing DRAM idle power is difficult because the volatile contents of the DRAM need to be refreshed (thereby consuming power) even when idle to ensure data availability. There have been a few approaches that propose reducing DRAM idle power by selectively refreshing only the important data [94, 123], or lowering the refresh rate [181, 122]. Recently, alternatives to DRAM memory, such as Flash memory [99] and Phase-Change Memory (PCM) [112], have also been proposed (and deployed). These technologies are non-volatile, and thus do not require much power when idle.

Reducing the idle power of disks is difficult because disks continue spinning, even when idle, to provide quick access for new data requests. Recent work [184, 200, 193] has proposed using a log data structure in combination with redundant disk systems, such as RAID, to allow some disks to spin down, thereby lowering idle power. Similar approaches have also been proposed [170, 14] for managing replicated data in distributed storage systems. There has also been significant orthogonal work trying to maximize server idle periods so as to facilitate the use of idle states [166, 15].

While lower idle power mitigates the need for dynamic server provisioning, our experiments in Section 4.6 show that the server idle power needs to be significantly low (less than 15W) for dynamic server provisioning to be rendered obsolete.

8.1.2 Voltage and frequency scaling

Dynamic Voltage and Frequency Scaling (DVFS) reduces active server power at the expense of lower processor speed (lower operating voltage and frequency). DVFS can be very useful in reducing server power in the typical operating range of 10% – 50% utilization [26]. DVFS has been successfully deployed in processors to lower server power consumption [69, 15, 151] in cases where the CPU is not required to operate at 100% speed. However, similar approaches have not yet been deployed for other server components, despite interest from the research community for DVFS in the memory subsystem (see, for example, [46, 50]), and for multi-speed disks for the storage subsystem (see, for example, [200, 201]).

Our own experience with DVFS [70, 69] has been positive. DVFS is a very attractive and practical solution since the transition time between DVFS states is negligible (on the order of microseconds [43]). However, the limited dynamic range of DVFS states, coupled with the fact that DVFS currently only applies to the CPU, limits the potential power savings obtained by DVFS.

8.2 Energy-efficient Server Design

A fundamental approach to power management is to design energy-efficient servers. There has been a lot of recent work in computer architecture that proposes novel, energy-efficient server designs.

8.2.1 Low-power inactive states

Anagnostopoulou et al. [17] and Amur et al. [16] propose low-power server states that provide access to the memory contents while sleeping. Such states are beneficial for servers that must provide access to their data contents at all times. Agarwal et al. [9, 10] propose a low-power inactive state that provides network connectivity even when asleep. This is beneficial for applications such as remote desktop. Reich et al. [156], Meisner et al. [134], and Nedevschi et al. [142], propose sleep states that lower setup times by building a sleep proxy, or a co-processor, that selectively wakes up the server based on incoming network traffic.

8.2.2 Heterogenous designs

There has been significant recent work proposing heterogeneity at the server level [195], processor level [82], and even the core level [49]. The basic idea here is to offer two modes of execution, one, which provides high performance and consumes high power, and the other, which provides lower performance and consumes less power. By switching to the most energy-efficient mode based on the application needs, overall server power can be reduced. This also helps servers achieve (some) power-proportionality.

Heterogeneity has also been exploited at the cluster level (see, for example, [107, 140]) to allocate workload to the most energy-efficient server architecture. Approaches leveraging heterogeneity are complementary to dynamic server provisioning, and typically provide moderate improvements over dynamic server provisioning.

8.2.3 Energy-efficient cluster architectures

Andersen et al. [19, 180] propose a low-power cluster design which makes use of embedded CPUs and flash storage to provide significant improvement in Performance-per-Watt for data-intensive applications. Similar architectures were also proposed

by Szalay et al. [169] and Caulfield et al. [37]. There have also been architectures that propose closely coupling processors with subsystems such as memory [118] and disk [158] to provide improved energy efficiency by reducing data transfers. Finally, there are also approaches that allocate workloads to the most efficient (existing) server architectures in a heterogeneous environment (see, for example, [107, 140]).

Interestingly, most of these approaches are complementary to dynamic server provisioning. Finding the right server design does not necessarily obviate the need for dynamic server provisioning. In fact, these approaches often require dynamic server provisioning, and also consolidation, so as to provide benefits in a distributed system. Most of the approaches discussed in this section propose designing a novel server architecture, and hence, apply to a single server. Dynamic server provisioning can build upon these approaches to provide an energy-efficient solution for a multi-server system. For example, dynamic server provisioning can leverage low-power inactive states such as those proposed in [142, 9, 17, 156, 16] to improve the energy efficiency of the system. As observed in our experiments in Section 4.5.1, there is a need for smart dynamic server provisioning even in the presence of inactive states with very low setup times. Setup times will have to approach zero before obviating dynamic server provisioning.

8.3 Consolidation and Virtualization

The basic idea behind consolidation is to colocate applications onto the same physical servers to amortize server power consumption. This "resource sharing" idea concentrates workload on fewer servers, thereby allowing unneeded servers to be turned off or repurposed. Virtualization [102, 25, 2] aids consolidation by making the application independent of its physical platform, thereby allowing application instances to easily migrate to other platforms. Most consolidation approaches leverage virtualization to consolidate application virtual machines (VMs) onto physical servers.

Several papers have addressed the allocation of workloads to physical servers via consolidation. Chase et al. [39] propose a decentralized bidding approach to consolidate application instances onto physical servers. Urgaonkar et al. [176] leverage over-subscription while consolidating VMs to improve resource utilization. Verma et al. [182] propose a theoretical framework to take migration costs into account when allocating applications to physical machines. Bobroff et al. [28] analyze workload demand traces to determine the best candidates for consolidation. Verma et al. [183]

also analyze workload traces, but they additionally take into account the correlation between different applications, to improve consolidation. Isci et al. [92] propose a remote direct memory access-based migration technique to reduce VM migration time, resulting in more aggressive consolidation.

There has also been significant work on minimizing the interference between colocated applications to avoid SLA violations. Nathuji et al. [141] propose a reactive feedback mechanism to provision additional (under-utilized) resources to VMs, thereby improving performance. Koh et al. [105] analyze colocated application VMs, and classify them into clusters based on their performance interference. Their analysis helps to provide performance isolation between VMs, thereby reducing SLA violations. Govindan et al. [80] examine cache interference between applications, and use their analysis to improve placement decisions.

Dynamic server provisioning can benefit from consolidation and virtualization. As such, these are complementary approaches. In fact, we use the ideas of consolidation in the design of *AutoScale* (see Chapter 4) when packing requests onto servers. However, consolidation is used at the process level in *AutoScale*, as opposed to being used at the application level. Extending *AutoScale* to deal with workload demand for multiple applications will require ideas from consolidation and virtualization, and is an interesting topic for future research.

8.4 Other Approaches

Apart from the approaches listed above, there are other approaches to indirectly reduce data center power consumption. These include: (i) cooling and thermal management (see, for example, [154, 84]), (ii) over-subscribing resources (see, for example, [57, 155]), (iii) coordinating power management approaches (see, for example, [153, 41, 45]), and (iv) using alternate energy sources to reduce grid power consumption (see, for example, [79, 78, 75, 149, 157]). These are important approaches to data center power management, and are complementary to dynamic server provisioning.

Chapter 9

Conclusion

Dynamic server provisioning has emerged as a promising approach for data center power management. The last decade has witnessed a lot of research activity in this area. Unfortunately, data center operators have not shown as much interest in employing dynamic server provisioning, since there are significant practical challenges that hinder its deployment in real data centers. Our goal in this thesis is to address some of the important challenges faced by dynamic server provisioning, and to provide practical solutions that enable it to be a viable deployment option in data centers. We now summarize the contributions made by this thesis, and list the future work opportunities that arise based on our work.

9.1 Contributions

9.1.1 Analyzing the challenges in dynamic provisioning

We explored the challenges that hinder the deployment of dynamic server provisioning in data centers. In particular, we examined setup costs, unpredictability in workload demand, and data availability in stateful servers (Chapter 3).

For setup costs, we provided the first closed-form analysis of multi-server systems with setup times (see Section 9.1.5 below). Our analysis revealed that setup costs in today's servers severely impact response time *and* power consumption (Section 3.1). Setup costs are incurred as a result of short-term fluctuations in load, which in turn necessitate dynamic changes in capacity. Fortunately, the adverse effects of setup costs decrease in severity as the system size scales up (Section 3.1.2). This suggests that dynamic server provisioning should be more beneficial for large data centers.

We then analyzed workload demand from data center application traces and publicly available web traces (Section 3.2). Our analysis revealed that most workloads are plagued by short-term fluctuations. Specifically, we found that while workload demands typically exhibit a predictable long-term pattern, they also exhibit unpredictable short-term fluctuations. Despite accounting for only a fraction of the total demand, these short-term fluctuations can lead to numerous SLA violations (Section 3.2.1). We also found that web traces additionally exhibit load spikes, which lead to a steep rise in response times in only a matter of seconds (Section 3.2.2).

Finally, we analyzed the challenges in dynamic provisioning of stateful servers (Section 3.3). Our experimental analysis revealed that adding or removing a node from the caching tier leads to temporary unavailability of cached data, resulting in a steep rise in response times (Section 3.3.1). This observation is probably the reason why there has been no prior work on scaling of the caching tier. Despite the challenges, there is significant potential for savings in the caching tier, since caching tier servers are typically equipped with lots of expensive and power-hungry DRAM.

9.1.2 AutoScale: Overcoming the challenges presented by setup costs and unpredictable workload demand

In Chapter 4, we designed and implemented a new dynamic server provisioning policy, *AutoScale*, that successfully addresses the challenges presented by setup costs and unpredictable workload demand. *AutoScale* demonstrates that in order to reduce power consumption without violating response time SLAs, it suffices to simply be conservative when scaling down capacity. Further, by monitoring the number of requests in the system, as opposed to request rate or server utilization, dynamic provisioning can be made robust to unpredictable changes in not just request rate, but also request size and server speeds. We also analyzed [64] the performance of *AutoScale* using theoretical modeling, and showed that, under more restrictive M/M/k models, *AutoScale* is near-optimal.

9.1.3 SoftScale: A new approach to handling load spikes

In Chapter 5, we proposed a novel approach, *SoftScale*, to handling load spikes in multi-tier data centers. *SoftScale* works by opportunistically stealing spare resources from other tiers to alleviate load in the bottleneck tier. We demonstrate that *SoftScale* can successfully meet response time SLAs, without consuming additional

resources, in the presence of severe load spikes caused by a sudden rise in request rate, or a sudden loss in system capacity. Importantly, *SoftScale* can be easily integrated with existing dynamic server provisioning policies to make them robust to load spikes.

9.1.4 CacheScale: Dynamically scaling the caching tier

In Chapter 6, we proposed a novel scaling policy, *CacheScale*, for dynamically provisioning resources in the stateful caching tier. We first made the case that there is a significant potential for savings in cost and power from dynamically provisioning the caching tier. We then showed that a small decrease in the required hit rate of the caching tier can lead to a significant reduction in the size of the caching tier. We then experimentally demonstrated that *CacheScale*, a simple data redistribution policy, allowed the caching tier to dynamically scale up and scale down in response to changes in load, without violating response time SLAs. *CacheScale* works by redistributing "hot" data items prior to scaling the caching tier. Interestingly, *CacheScale* does not require knowledge of historical data accesses in the cache. This makes it very easy to deploy *CacheScale* on existing caching tiers.

9.1.5 Recursive Renewal Reward: A new technique for solving Markov chains with a repeating structure

In Chapter 7, we presented a new analytical technique, Recursive Renewal Reward (RRR), for analyzing Markov chains with a repeating structure. Such Markov chains are widely used to model computer systems, inventory control systems, production systems, and call center staffing systems. In particular, multi-server computer systems with setup times are often modeled using an M/M/k/setup Markov chain, which is a 2-dimensional Markov chain with an infinite repeating portion. Given the complexity of the M/M/k/setup chain, its analysis has eluded researchers for many decades: While the M/M/1/setup was exactly analyzed in 1964, no exact analysis exists to date for the M/M/k/setup with $k > 1$. Using RRR, we provided the first exact, closed-form analysis for the M/M/k/setup. The resulting analysis provided interesting insights into the impact of setup time on response time and power consumption in data centers (see Section 3.1).

RRR uses ideas from renewal reward theory and busy period analysis to obtain closed-form expressions for metrics of interest such as the transform of time in

system and the transform of power consumed by the system. The simplicity, intuitiveness, and versatility of RRR makes it useful for analyzing Markov chains far beyond the M/M/k/setup. In general, RRR should be able to reduce the analysis of any 2-dimensional Markov chain which is finite in one dimension, say the vertical dimension, and infinite (with repeating structure) in the other (horizontal dimension) to the problem of solving a system of polynomial equations. Further, if in the repeating portion all horizontal transitions are skip-free and all vertical transitions are unidirectional, the resulting system of equations will be at most quadratic, yielding a closed-form solution. We thus anticipate that RRR will prove useful to other researchers in analyzing many new problems.

9.2 Future Work

This thesis proposed several dynamic server provisioning policies for data center power management. There are two immediate extensions to our work that we did not address in this thesis. The first is the extension of our dynamic server provisioning policies to virtual environments. That is, how would our proposed policies change if our implementation testbed was instantiated on VMs as opposed to physical servers. In a virtualized environment, the goal is to save on rental costs by dynamically scaling the number of VMs, as opposed to reducing power consumption by dynamically provisioning physical servers. The potentially lower setup time for VMs [108, 13] should allow for more aggressive scaling. However, the setup time for a VM can vary significantly depending on external factors such as time of day, location of host data center, etc [127]. This variation in setup time could influence the design of the dynamic provisioning policy.

The second immediate extension that this thesis does not address is dynamic provisioning in the presence of heterogeneous resources. There is often heterogeneity in physical servers [140] and VMs [127]. Prior studies indicate that leveraging this heterogeneity can be beneficial for resource management [140, 107]. It will be interesting to adapt our dynamic server provisioning policies to heterogeneous environments, and to examine the subsequent benefits.

Our dynamic server provisioning policies in this thesis have been proposed and evaluated in the context of our implementation testbed that mimics a multi-tier web application. We have not considered dynamic provisioning for other applications, such as data-intensive applications or high-performance computing applications, or dynamic provisioning for a mix of applications. These are important questions, and are left as future work.

Finally, an important question that we partially addressed (see Section 3.1.2) in this thesis is the effect of scale on dynamic server provisioning. Our preliminary findings indicate that dynamic provisioning is more beneficial for large-scale data centers. This is an encouraging result, and warrants further research.

Bibliography

[1] libMemcached. http://libmemcached.org/libMemcached.html. 6.4

[2] VMware Virtualization Technology. http://www.vmware.com. 8.3

[3] Tarek Abdelzaher, Yixin Diao, Joseph L. Hellerstein, Chenyang Lu, and Xiaoyun Zhu. Introduction to control theory and its application to computing systems. In *Performance Modeling and Engineering*, chapter 7, pages 185–215. Springer US, 2008. 2.1.2, 2.2, 2.2.2

[4] I. Adan and J Resing. A class of Markov processes on a semi-infinite strip. Technical Report 99-03, Eindhoven University of Technology, Department of Mathematics and Computing Sciences, 1999. 7.2.2

[5] I. Adan and J. van der Wal. Combining make to order and make to stock. *OR Spektrum*, 20:73–81, 1998. 7.2.4

[6] I. Adan and J. van der Wal. Combining make to order and make to stock. *OR Spektrum*, 20:73–81, 1998. 2.2.6

[7] Stephen Adler. The Slashdot Effect: An Analysis of Three Internet Publications. http://ssadler.phy.bnl.gov/adler/SDE/SlashDotEffect.html, 2005. 1.3.3, 3.2.2, 5.1

[8] Atul Adya, William J. Bolosky, Ronnie Chaiken, John R. Douceur, Jon Howell, and Jacob Lorch. Load Management in a Large-Scale Decentralized File System. Technical Report MSR-TR-2004-60, Microsoft Research, July 2004. 5.7

[9] Yuvraj Agarwal, Steve Hodges, Ranveer Chandra, James Scott, Paramvir Bahl, and Rajesh Gupta. Somniloquy: Augmenting network interfaces to reduce PC energy usage. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '09, pages 365–380, Boston, MA, USA, 2009. 8.2.1, 8.2.3

[10] Yuvraj Agarwal, Stefan Savage, and Rajesh Gupta. SleepServer: A software-only approach for reducing the energy consumption of PCs within enterprise environments. In *Proceedings of the 2010 USENIX Annual Technical Confer-*

ence, ATC '10, pages 285–299, Boston, MA, USA, 2010. 8.2.1

[11] U.S. Environmental Protection Agency. EPA Report on Server and Data Center Energy Efficiency. 2007. 1.2, 1.2.2

[12] Virgílio Almeida, Azer Bestavros, Mark Crovella, and Adriana de Oliveira. Characterizing reference locality in the WWW. In *Proceedings of the 4th International Conference on Parallel and Distributed Information Systems*, DIS '96, pages 92–107, Miami Beach, FL, USA, 1996. 6.1, 6.3.1, 6.3.3

[13] Amazon Inc. Amazon Elastic Compute Cloud (Amazon EC2). `http://aws.amazon.com/ec2`, 2008. 1.1, 1.3.3, 2.1, 3.2.2, 4.1, 5.1, 6.1, 1, 6.3.3, 9.2

[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 '10, pages 217–228, Indianapolis, IN, USA, 2010. 2.1.2, 2.2.4, 5.8, 6.1, 8.1.1

[15] Hrishikesh Amur, Ripal Nathuji, Mrinmoy Ghosh, Karsten Schwan, and Hsien-Hsin S. Lee. Idlepower: Application-Aware Management of Processor Idle States. In *Proceedings of the Workshop on Managed Many-Core Systems*, MMCS '08, Boston, MA, USA, 2008. 1.3.1, 8.1.1, 8.1.2

[16] Hrishikesh Amur and Karsten Schwan. Achieving power-efficiency in clusters without distributed file system complexity. In *Workshop on Energy Efficient Design*, WEED '10, Saint-Malo, France, 2010. 8.2.1, 8.2.3

[17] Vlasia Anagnostopoulou, Susmit Biswas, Heba Saadeldeen, Alan Savage, Ricardo Bianchini, Tao Yang, Diana Franklin, and Frederic T. Chong. Barely alive memory servers: Keeping data active in a low-power state. *Journal on Emerging Technologies in Computing Systems*, 8(4):1–20, November 2012. 8.2.1, 8.2.3

[18] David G. Andersen. Trace of web site activity on Pi day (3/14/2011) from domains hosted by angio.net. Personal communication, December 2011. 5.4.2, 5.10

[19] David G. Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. FAWN: a fast array of wimpy nodes. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, SOSP '09, pages 1–14, Big Sky, MT, USA, 2009. 8.2.3

[20] Martin Arlitt and Tai Jin. Workload Characterization of the 1998 World Cup Web Site. *IEEE Network*, 14:30–37, 2000. 1.3.3, 3.2.2, 5.1, 5.4.2, 5.8

[21] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andrew Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. Above the clouds: A Berkeley view of cloud computing. Technical Report UCB/EECS-2009-28, EECS Department, University

of California, Berkeley, 2009. 2.1.1, 4.1

[22] J. R. Artalejo, A. Economou, and M. J. Lopez-Herrero. Analysis of a Multi-server Queue with Setup Times. *Queueing Systems: Theory and Applications*, 51(1-2):53–76, 2005. 2.2.6

[23] J. R. Artalejo, A. Economou, and M. J. Lopez-Herrero. Analysis of a multi-server queue with setup times. *Queueing Syst. Theory Appl.*, 51(1-2):53–76, 2005. 7.2.4

[24] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload Analysis of a Large-Scale Key-Value Store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, pages 53–64, London, UK, 2012. 5.1, 5.1, 5.8

[25] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, SOSP '03, pages 164–177, Bolton Landing, NY, USA, 2003. 8.3

[26] Luiz André Barroso and Urs Hölzle. The Case for Energy-Proportional Computing. *IEEE Computer*, 40(12):33–37, 2007. 1, 1.3.1, 2.1.1, 2.1.3, 3.1, 3.1.2, 4.1, 7.1, 8.1, 8.1.2

[27] Doug Beaver, Sanjeev Kumar, Harry C. Li, Jason Sobel, and Peter Vajgel. Finding a needle in Haystack: Facebook's photo storage. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI '10, pages 47–60, Vancouver, British Columbia, Canada, 2010. 6.3.3

[28] N. Bobroff, A. Kochut, and K. Beaty. Dynamic placement of virtual machines for managing SLA violations. In *Proceedings of the 10th IFIP/IEEE International Symposium on Integrated Network Management*, IM '07, pages 119–128, Munich, Germany, 2007. 2.2, 4.1, 5.8, 8.3

[29] Peter Bodik, Michael Paul Armbrust, Kevin Canini, Armando Fox, Michael Jordan, and David A. Patterson. A Case for Adaptive Datacenters to Conserve Energy and Improve Reliability. Technical Report UCB/EECS-2008-127, EECS Department, University of California, Berkeley, 2008. 4.4

[30] Peter Bodík, Rean Griffith, Charles Sutton, Armando Fox, Michael Jordan, and David Patterson. Statistical machine learning makes automatic control practical for internet datacenters. In *Proceedings of the 2009 Conference on Hot Topics in Cloud Computing*, HotCloud '09, San Diego, CA, USA, 2009. 4.3.4

[31] L. Breslau, Pei Cao, Li Fan, G. Phillips, and S. Shenker. Web caching and Zipf-like distributions: evidence and implications. In *Proceedings of 1999 IEEE*

INFOCOM, INFOCOM '99, pages 126–134, New York, NY, USA, 1999. 6.1, 6.3.1, 6.3.1, 6.3.3

[32] David R. Brillinger. *Time Series : Data Analysis and Theory*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2001. 3.2.1

[33] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual Web search engine. In *Proceedings of the 7th International Conference on World Wide Web*, WWW7, pages 107–117, Brisbane, Australia, 1998. 1.1

[34] Roy Bryant, Alexey Tumanov, Olga Irzak, Adin Scannell, Kaustubh Joshi, Matti Hiltunen, Andres Lagar-Cavilla, and Eyal de Lara. Kaleidoscope: cloud micro-elasticity via VM state coloring. In *Proceedings of the 6th European Conference on Computer Systems*, EuroSys '11, pages 273–286, Salzburg, Austria, 2011. 5.1, 5.1, 5.8

[35] Peter Burrows. Apple Says Data Centers Now Use 100% Renewable Energy. http://www.bloomberg.com/news/2013-03-21/apple-says-data-centers-now-use-100-renewable-energy.html, March 2013. 1.2.1

[36] Malu Castellanos, Fabio Casati, Ming-Chien Shan, and Umesh Dayal. iBOM: A Platform for Intelligent Business Operation Management. In *Proceedings of the 21st International Conference on Data Engineering*, ICDE '05, pages 1084–1095, Tokyo, Japan, 2005. 2.2, 4.1

[37] Adrian M. Caulfield, Laura M. Grupp, and Steven Swanson. Gordon: Using flash memory to build fast, power-efficient clusters for data-intensive applications. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '09, pages 217–228, Washington, DC, USA, 2009. 8.2.3

[38] Abhishek Chandra and Prashant Shenoy. Effectiveness of Dynamic Resource Allocation for Handling Internet Flash Crowds. Technical Report TR03-37, Department of Computer Science, University of Massachusetts at Amherst, November 2003. 5.7

[39] Jeffrey S. Chase, Darrell C. Anderson, Prachi N. Thakar, and Amin M. Vahdat. Managing energy and server resources in hosting centers. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, SOSP '01, pages 103–116, Lake Louise, Alberta, Canada, 2001. 2.2, 2.2.5, 8.3

[40] 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 '08, pages 337–350, San Francisco, CA, USA, 2008. 2.1.2, 2.2, 2.2.1, 4.1, 4.2.1, 4.3.1, 4.8,

6.1

[41] Yiyu Chen, Amitayu Das, Wubi Qin, Anand Sivasubramaniam, Qian Wang, and Natarajan Gautam. Managing Server Energy and Operational Costs in Hosting Centers. In *Proceedings of the 2005 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '05, pages 303–314, Banff, Alberta, Canada, 2005. 2.1.2, 2.2, 4.1, 4.7.4, 8.4

[42] Ludmila Cherkasova and Peter Phaal. Session-Based Admission Control: A Mechanism for Peak Load Management of Commercial Web Sites. *IEEE Transactions on Computers*, 51, June 2002. 5.7

[43] Kihwan Choi, Ramakrishna Soma, and Massoud Pedram. Fine-Grained Dynamic Voltage and Frequency Scaling for Precise Energy and Performance Trade-Off Based on the Ratio of Off-Chip Access to On-Chip Computation Times. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '04, pages 4–9, Paris, France, 2004. 1.3.1, 8.1.2

[44] Josh Constine. Walmarts Black Friday Disaster: Website Crippled, Violence In Stores. http://techcrunch.com/2011/11/25/walmart-black-friday, November 2011. 1.3.3, 3.2.2, 5.1, 5.8

[45] Rajarshi Das, Srinivas Yarlanki, Hendrik Hamann, Jeffrey O. Kephart, and Vanessa Lopez. A unified approach to coordinated energy-management in data centers. In *Proceedings of the 7th International Conference on Network and Services Management*, CNSM '11, pages 504–508, Paris, France, 2011. 8.4

[46] Howard David, Chris Fallin, Eugene Gorbatov, Ulf R. Hanebutte, and Onur Mutlu. Memory power management via dynamic voltage/frequency scaling. In *Proceedings of the 8th ACM International Conference on Autonomic Computing*, ICAC '11, pages 31–40, Karlsruhe, Germany, 2011. 1.3.1, 8.1.2

[47] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of ACM*, 51(1):107–113, January 2008. 1.1, 5.4.1

[48] 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 21st ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 205–220, Stevenson, WA, USA, 2007. 1.1, 1.1.1, 2.3.1, 2.3.2, 4.1, 4.2.1, 4.3, 6.2, 7.1

[49] Qingyuan Deng, David Meisner, Abhishek Bhattacharjee, Thomas F. Wenisch, and Ricardo Bianchini. CoScale: Coordinating CPU and Memory System DVFS in Server Systems. In *Proceedings of the 45th Annual IEEE/ACM In-*

ternational Symposium on Microarchitecture, MICRO '12, pages 143–154, Vancouver, British Columbia, Canada, 2012. 8.2.2

[50] Qingyuan Deng, David Meisner, Luiz Ramos, Thomas F. Wenisch, and Ricardo Bianchini. MemScale: active low-power modes for main memory. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '11, pages 225–238, Newport Beach, CA, USA, 2011. 1.3.1, 8.1.2

[51] V. Devadas and H. Aydin. On the Interplay of Voltage/Frequency Scaling and Device Power Management for Frame-based Real-Time Embedded Applications. *IEEE Transactions on Computers*, 61(1):31–44, January 2012. 4.5

[52] Wayne W. Eckerson. Three Tier Client/Server Architecture: Achieving Scalability, Performance, and Efficiency in Client Server Applications. *Open Information Systems*, 10(1), January 1995. 5.1, 5.8

[53] E.N. Elnozahy, Michael Kistler, and Ramakrishnan Rajamony. Energy-efficient server clusters. In *Proceedings of the 2nd Workshop on Power-Aware Computing Systems*, WPACS '02, pages 179–196, Cambridge, MA, USA, 2002. 2.1.2, 2.2, 4.1

[54] Jeremy Elson and Jon Howell. Handling flash crowds from your garage. In *Proceedings of the 2008 USENIX Annual Technical Conference*, ATC '08, pages 171–184, Boston, MA, USA, 2008. 5.7

[55] Facebook. Personal communication with Facebook, 2011. 2.3.2, 4.1, 4.2.1, 6.1

[56] Michalis Faloutsos, Petros Faloutsos, and Christos Faloutsos. On power-law relationships of the Internet topology. In *Proceedings of the ACM SIGCOMM 1999 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '99, pages 251–262, Cambridge, MA, USA, 1999. 6.3.1

[57] Xiaobo Fan, Wolf-Dietrich Weber, and Luiz André Barroso. Power provisioning for a warehouse-sized computer. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, pages 13–23, San Diego, CA, USA, 2007. 1.2.3, 2.1.2, 2.2, 4.1, 8.4

[58] Donald F. Ferguson, Christos Nikolaou, Jakka Sairamesh, and Yechiam Yemini. *Economic models for allocating resources in computer systems*, pages 156–183. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1996. 2.2, 2.2.5

[59] Brad Fitzpatrick. Distributed caching with memcached. *Linux Journal*, 2004(124), 2004. 1.1, 3.3.1, 6.1, 6.2

[60] A. Gandhi, Y. Chen, D. Gmach, M. Arlitt, and M. Marwah. Minimizing Data Center SLA Violations and Power Consumption via Hybrid Resource Provisioning. In *Proceedings of the 2011 International Green Computing Conference*,

IGCC '11, pages 49–56, Orlando, FL, USA, 2011. 2.1.2, 2.2, 2.2.3, 3.2.1, 3.2.1, 3.2.1, 4.1, 4.4, 5.1, 5.1

[61] A. Gandhi, Y. Chen, D. Gmach, M. Arlitt, and M. Marwah. Hybrid Resource Provisioning for Minimizing Data Center SLA Violations and Power Consumption. *Sustainable Computing: Informatics and Systems*, 2:91–104, 2012. 2.1.2, 2.2, 2.2.3, 3.2.1, 3.2.1, 4.1, 4.1(f), 4.3.6, 4.5.1

[62] Anshul Gandhi, Sherwin Doroudi, Mor Harchol-Balter, and Alan Scheller-Wolf. Exact Analysis of the M/M/k/setup Class of Markov Chains via Recursive Renewal Reward. In *Proceedings of the 2013 ACM International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '13, pages 153–166, Pittsburgh, PA, USA, 2013. 2.2.6, 7.4, 7.6, 7.6.1, 7.6.2, 7.6.3, 7.7

[63] Anshul Gandhi, Sherwin Doroudi, Mor Harchol-Balter, and Alan Scheller-Wolf. Exact Analysis of the M/M/k/setup Class of Markov Chains via Recursive Renewal Reward. Technical Report CMU-CS-13-105, Carnegie Mellon University, 2013. 7.4, 7.6, 7.6.3

[64] Anshul Gandhi, Varun Gupta, Mor Harchol-Balter, and Michael Kozuch. Optimality Analysis of Energy-Performance Trade-off for Server Farm Management. *Performance Evaluation*, 67:1155–1171, 2010. 2.3.1, 4.3.5, 9.1.2

[65] Anshul Gandhi and Mor Harchol-Balter. How Data Center Size Impacts the Effectiveness of Dynamic Power Management. pages 1164 – 1169, 2011. 7.8

[66] Anshul Gandhi, Mor Harchol-Balter, and Ivo Adan. Analysis of an M/M/k system with exponential setup times under staggered boot up. In *Madrid Conference on Queueing Theory*, 2010. 2.2.6

[67] Anshul Gandhi, Mor Harchol-Balter, and Ivo Adan. Decomposition results for an M/M/k with Staggered Setup. *Performance Evaluation Review*, 38:48–50, 2010. 2.2.6

[68] Anshul Gandhi, Mor Harchol-Balter, and Ivo Adan. Server farms with setup costs. *Performance Evaluation*, 67:1123–1138, 2010. 1.3.3, 2.2.6, 7.1, 7.2.4, 7.8

[69] Anshul Gandhi, Mor Harchol-Balter, Rajarshi Das, Jeffrey Kephart, and Charles Lefurgy. Power Capping Via Forced Idleness. In *Workshop on Energy Efficient Design*, WEED '09, Austin, TX, USA, 2009. 1.3.1, 4.6, 8.1.2

[70] Anshul Gandhi, Mor Harchol-Balter, Rajarshi Das, and Charles Lefurgy. Optimal power allocation in server farms. In *Proceedings of the 11th International Joint Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '09, pages 157–168, Seattle, WA, USA, 2009. 1.3.1, 8.1.2

[71] Anshul Gandhi, Mor Harchol-Balter, and Michael Kozuch. The case for sleep states in servers. In *Proceedings of the 4th Workshop on Power-Aware Computing and Systems*, HotPower '11, Cascais, Portugal, 2011. 4.5.3

[72] Anshul Gandhi, Mor Harchol-Balter, and Michael Kozuch. Are Sleep States Effective in Data Centers? In *Proceedings of the 2012 International Green Computing Conference*, IGCC '12, pages 113–122, San Jose, CA, USA, 2012. 2.3.1, 4.5.3

[73] Anshul Gandhi, Mor Harchol-Balter, Ram Raghunathan, and Michael Kozuch. AutoScale: Dynamic, Robust Capacity Management for Multi-Tier Data Centers. *Transactions on Computer Systems*, 30, 2012. 2.1.2

[74] N. Gandhi, D.M. Tilbury, Y. Diao, J. Hellerstein, and S. Parekh. MIMO control of an Apache web server: Modeling and controller design. In *Proceedings of the 2002 American Control Conference*, volume 6 of *ACC '02*, pages 4922 – 4927, Anchorage, AK, USA, 2002. 4.7.4

[75] D. Gmach, J. Rolia, C. Bash, Yuan Chen, T. Christian, A. Shah, R. Sharma, and Zhikui Wang. Capacity planning and power management to exploit sustainable energy. In *Proceedings of the 2010 International Conference on Network and Service Management*, CSNM '10, pages 96–103, Niagara Falls, Ontario, Canada, 2010. 8.4

[76] Daniel Gmach, Stefan Krompass, Andreas Scholz, Martin Wimmer, and Alfons Kemper. Adaptive quality of service management for enterprise services. *ACM Transactions on the Web*, 2:1–46, 2008. 2.2, 4.1, 5.1, 5.1

[77] Daniel Gmach, Jerry Rolia, Ludmila Cherkasova, and Alfons Kemper. Resource pool management: Reactive versus proactive or let's be friends. *Computer Networks*, 53(17):2905–2922, December 2009. 1.3.3

[78] Íñigo Goiri, William Katsak, Kien Le, Thu D. Nguyen, and Ricardo Bianchini. Parasol and GreenSwitch: Managing datacenters powered by renewable energy. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 51–64, Houston, TX, USA, 2013. 8.4

[79] Íñigo Goiri, Kien Le, Thu D. Nguyen, Jordi Guitart, Jordi Torres, and Ricardo Bianchini. GreenHadoop: leveraging green energy in data-processing frameworks. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 57–70, Bern, Switzerland, 2012. 8.4

[80] 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*, SOCC '11, pages 1–14, Cascais, Portugal, 2011. 5.8, 8.3

[81] Dirk Grunwald, Charles B. Morrey, III, Philip Levis, Michael Neufeld, and Keith I. Farkas. Policies for dynamic clock scheduling. In *Proceedings of the 4th Conference on Symposium on Operating System Design and Implementation*,

OSDI '00, San Diego, CA, USA, 2000. 4.3.4

[82] Vishal Gupta, Paul Brett, David Koufaty, Dheeraj Reddy, Scott Hahn, Karsten Schwan, and Ganapati Srinivasa. HeteroMates: Providing high dynamic power range on client devices using heterogeneous core groups. In *Proceedings of the 2012 International Green Computing Conference*, IGCC '12, pages 1–10, San Jose, CA, USA, 2012. 8.2.2

[83] Dan Haugen. How wind energy helped Iowa attract Facebooks new data center. http://www.midwestenergynews.com/2013/04/24/how-wind-energy-helped-iowa-attract-facebooks-new-data-center, April 2013. 1.2.1

[84] Taliver Heath, Ana Paula Centeno, Pradeep George, Luiz Ramos, Yogesh Jaluria, and Ricardo Bianchini. Mercury and Freon: Temperature Emulation and Management for Server Systems. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '06, pages 106–116, San Jose, CA, USA, 2006. 8.4

[85] Henry Hoffmann, Stelios Sidiroglou, Michael Carbin, Sasa Misailovic, Anant Agarwal, and Martin Rinard. Dynamic knobs for responsive power-aware computing. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '11, pages 199–212, Newport Beach, CA, USA, 2011. 4.8

[86] Tibor Horvath and Kevin Skadron. Multi-mode energy management for multi-tier server clusters. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, pages 270–279, Toronto, Ontario, Canada, 2008. 2.1.2, 2.1.3, 2.2, 2.2.2, 4.1, 4.3.1, 4.5, 4.7.4, 5.1, 5.1

[87] Chung-hsing Hsu and Wu-chun Feng. A power-aware run-time system for high-performance computing. In *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, SC '05, Seattle, WA, USA, 2005. 1.3.1

[88] Jim Hu and Greg Sandoval. Web acts as hub for info on attacks. *CNET news*, September 2001. 1.3.3, 3.2.2, 5.1, 5.8

[89] Intel Corp. Intel® Core™2 Duo Mobile Processor Datasheet: Table 20. http://download.intel.com/design/mobile/datashts/32012001.pdf, 2008. 1.3.1, 8.1.1

[90] Sandy Irani, Sandeep Shukla, and Rajesh Gupta. Algorithms for power savings. *ACM Transactions on Algorithms*, 3(4), November 2007. 2.2, 2.2.6

[91] David Irwin, Jeffrey Chase, Laura Grit, Aydan Yumerefendi, David Becker, and Kenneth G. Yocum. Sharing networked resources with brokered leases. In *Proceedings of the 2006 USENIX Annual Technical Conference*, ATC '06,

pages 199–212, Boston, MA, USA, 2006. 2.2, 2.2.5

[92] C. Isci, J. Liu, B. Abali, J.O. Kephart, and J. Kouloheris. Improving server utilization using fast virtual machine migration. *IBM Journal of Research and Development*, 55(6):365–376, 2011. 8.3

[93] Canturk Isci, Suzanne McIntosh, Jeffrey O. Kephart, Rajarshi Das, James Hanson, Scott Piper, Robert Wolford, Tom Brey, Robert Kanter, Allen Ng, James Norris, Abdoulaye Traore, and Michael Frissora. Agile, Efficient Virtualization Power Management with Low-latency Server Power States. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, 2013. 1.3.3, 2.1.2, 3.1, 3.2.2, 4.1, 4.9, 5.1

[94] Ciji Isen and Lizy John. ESKIMO: Energy savings using Semantic Knowledge of Inconsequential Memory Occupancy for DRAM subsystem. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '09, pages 337–346, New York, NY, USA, 2009. 8.1.1

[95] ITA. The Internet Traffic Archives: WorldCup98. http://ita.ee.lbl.gov/html/contrib/WorldCup.html, 1998. 3.2.1, 4.1(a), 4.3.6, 4.7.3, 4.4, 4.7.3

[96] Sitaram Iyer and Peter Druschel. Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous I/O. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, SOSP '01, pages 117–130, Banff, Alberta, Canada, 2001. 4.3.5

[97] Otis B. Jennings, Avishai M, William A. Massey, and Ward Whitt. Server Staffing to Meet Time-Varying Demand. *Management Science*, 42:1383–1394, 1996. 2.2.6, 4.3.5

[98] J Keilson and L.D Servi. A distributional form of little's law. *Operations Research Letters*, 7(5):223 – 227, 1988. 7.5.4

[99] Taeho Kgil and Trevor Mudge. FlashCache: A NAND flash memory file cache for low power web servers. In *Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, CASES '06, pages 103–112, Seoul, Korea, 2006. 8.1.1

[100] J. Kim and T. S. Rosing. Power-aware resource management techniques for low-power embedded systems. In S. H. Son, I. Lee, and J. Y-T Leung, editors, *Handbook of Real-Time and Embedded Systems*. Taylor-Francis Group LLC, 2006. 4.3.5

[101] Yoongu Kim, Vivek Seshadri, Donghyuk Lee, Jamie Liu, and Onur Mutlu. A case for exploiting subarray-level parallelism (SALP) in DRAM. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, pages 368–379, Portland, OR, USA, 2012. 5.6

[102] Avi Kivity. kvm: the Linux virtual machine monitor. In *Proceedings of the*

2007 Ottawa Linux Symposium, OLS '07, pages 225–230, Ottawa, Ontario, Canada, 2007. 1.3.3, 3.2.2, 4.1, 5.1, 8.3

[103] Leonard Kleinrock. *Queueing Systems, Volume I: Theory*. Wiley-Interscience, 1975. 4.7.3, 4.7.4, 6.3.2, 7.5.3

[104] Data Center Knowledge. "Who Has the Most Web Servers?". http://www.datacenterknowledge.com/archives/2009/05/14/whos-got-the-most-web-servers, 2009. 1.2

[105] Younggyun Koh, R. Knauerhase, P. Brett, M. Bowman, Zhihua Wen, and C. Pu. An Analysis of Performance Interference Effects in Virtual Environments. In *Proceedings of the 2007 IEEE International Symposium on Performance Analysis of Systems Software*, ISPASS '07, pages 200–209, San Jose, CA, USA, 2007. 5.8, 8.3

[106] Vasileios Kontorinis, Liuyi Eric Zhang, Baris Aksanli, Jack Sampson, Houman Homayoun, Eddie Pettis, Dean M. Tullsen, and Tajana Simunic Rosing. Managing distributed ups energy for effective power capping in data centers. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, pages 488–499, Portland, OR, USA, 2012. 1.2.3

[107] Andrew Krioukov, Prashanth Mohan, Sara Alspaugh, Laura Keys, David Culler, and Randy Katz. NapSAC: Design and implementation of a power-proportional web cluster. In *Proceedings of the 1st ACM SIGCOMM Workshop on Green Networking*, Green Networking '10, pages 15–22, New Delhi, India, 2010. 2.1.2, 2.2, 2.2.1, 2.3.1, 4.1, 4.3, 4.3.1, 4.3.4, 4.5, 4.8, 5.1, 5.1, 6.1, 7.1, 8.2.2, 8.2.3, 9.2

[108] Horacio Andrés Lagar-Cavilla, Joseph Andrew Whitney, Adin Matthew Scannell, Philip Patchin, Stephen M. Rumble, Eyal de Lara, Michael Brudno, and Mahadev Satyanarayanan. Snowflock: rapid virtual machine cloning for cloud computing. In *Proceedings of the 4th ACM European Conference on Computer Systems*, EuroSys '09, pages 1–12, Nuremberg, Germany, 2009. 1.3.3, 2.1, 3.2.2, 4.1, 5.1, 9.2

[109] Petter Larsson. Power Efficiency Analysis and SW Development Recommendations for Intel Atom based MID platforms. Intel Software & Services Group White Paper, March 2009. 1.3.1, 8.1.1

[110] Ed Lassettre, David W. Coleman, Yixin Diao, Steve Froehlich, Joseph L. Hellerstein, Lawrence S. Hsiung, Todd W. Mummert, Mukund Raghavachari, Geoffrey Parker, Lance Russell, Maheswaran Surendra, Veronica Tseng, Noshir Wadia, and Pery Ye. Dynamic surge protection: An approach to handling unexpected workload surges with resource actions that have lead times. In *Proceedings of the 14th IFIP/IEEE International Workshop on Distributed Systems:*

Operations and Management, DSOM '03, pages 82–92, Heidelberg, Germany, 2003. 5.7

[111] G. Latouche and V. Ramaswami. *Introduction to Matrix Analytic Methods in Stochastic Modeling*. ASA-SIAM, Philadelphia, 1999. 7.1, 7.2.1, 2

[112] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Architecting phase change memory as a scalable dram alternative. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 2–13, Austin, TX, USA, 2009. 8.1.1

[113] William LeFebvre. CNN.com: Facing A World Crisis. *Invited Talk, USENIX Annual Technical Conference*, June 2002. 1.3.3, 3.2.2, 5.1, 5.8

[114] Julius C.B. Leite, Dara M. Kusic, and Daniel Mossé. Stochastic approximation control of power and tardiness in a three-tier web-hosting cluster. In *Proceeding of the 7th International Conference on Autonomic Computing*, ICAC '10, pages 41–50, Washington, DC, USA, 2010. 2.1.2, 2.2, 4.1, 4.7.4, 5.1, 5.1

[115] Jacob Leverich and Christos Kozyrakis. On the energy (in)efficiency of Hadoop clusters. *SIGOPS Operating Systems Review*, 44(1):61–65, March 2010. 2.2.4

[116] Y. Levy and U. Yechiali. An M/M/s queue with servers' vacations. *Canadian Journal of Operational Research and Information Processing*, 14(2):153–163, 1976. 7.2.3

[117] Baochun Li and Klara Nahrstedt. A Control-Based Middleware Framework for Quality of Service Adaptations. *IEEE Journal on Selected Areas in Communications*, 17:1632–1650, 1999. 4.7.4

[118] Kevin Lim, Parthasarathy Ranganathan, Jichuan Chang, Chandrakant Patel, Trevor Mudge, and Steven Reinhardt. Understanding and Designing New Server Architectures for Emerging Warehouse-Computing Environments. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, pages 315–326, Beijing, China, 2008. 8.2.3

[119] Seung-Hwan Lim, Bikash Sharma, Byung Chul Tak, and Chita R. Das. A dynamic energy management scheme for multi-tier data centers. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS '11, pages 257–266, Austin, TX, USA, 2011. 2.2

[120] Minghong Lin, Zhenhua Liu, Adam Wierman, and Lachlan Andrew. Online algorithms for geographical load balancing. In *Proceedings of the 2012 International Green Computing Conference*, IGCC '12, pages 166–175, San Jose, CA, USA, 2012. 2.1.2, 2.2, 2.2.6, 4.4

[121] Minghong Lin, Adam Wierman, Lachlan Andrew, and Eno Thereska. Dynamic right-sizing for power-proportional data centers. In *Proceedings of 2011 IEEE INFOCOM*, INFOCOM '11, pages 1098–1106, Shanghai, China, 2011. 2.1.2,

2.2, 2.2.6, 4.4

[122] Jamie Liu, Ben Jaiyen, Richard Veras, and Onur Mutlu. RAIDR: Retention-Aware Intelligent DRAM Refresh. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, pages 1–12, Portland, OR, USA, 2012. 8.1.1

[123] Song Liu, Karthik Pattabiraman, Thomas Moscibroda, and Benjamin G. Zorn. Flikker: saving DRAM refresh-power through critical data partitioning. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '11, pages 213–224, Newport Beach, CA, USA, 2011. 8.1.1

[124] C. Lu, Y. Lu, T.F. Abdelzaher, J.A. Stankovic, and S.H. Son. Feedback Control Architecture and Design Methodology for Service Delay Guarantees in Web Servers. *IEEE Transactions on Parallel and Distributed Systems*, 17(9):1014–1027, September 2006. 4.7.4

[125] Yung-Hsiang Lu, Eui-Young Chung, Tajana Šimunić, Luca Benini, and Giovanni De Micheli. Quantitative comparison of power management algorithms. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '00, pages 20–26, Paris, France, 2000. 4.3.5, 4.5

[126] Mohamed S. Mansour, Karsten Schwan, and Sameh Abdelaziz. Isolation points: Creating performance-robust enterprise systems. *Transactions on Autonomous and Adaptive Systems*, 4(2):1–28, May 2009. 5.4.1

[127] Ming Mao and Marty Humphrey. A Performance Study on the VM Startup Time in the Cloud. In *Proceedings of the 5th IEEE International Conference on Cloud Computing*, CLOUD '12, pages 423–430, Honolulu, HI, USA, 2012. 1.3.3, 9.2

[128] Mark LaPedus. Facebook Wants New and Cheaper Memories. http://semimd.com/blog/2011/11/08/facebook-wants-new-and-cheaper-memories, November 2011. 2.1.3, 3.3, 5.2.2, 6, 6.1

[129] Vimal Mathew, Ramesh Sitaraman, and Prashant Shenoy. Energy-aware load balancing in content delivery networks. In *Proceedings of 2012 IEEE INFOCOM*, INFOCOM '12, pages 954–962, Orlando, FL, USA, 2012. 1.3.3, 2.1.2, 3.1, 3.2.2, 4.1, 4.4, 4.9, 5.1

[130] D. R. Mauro and K. J. Schmidt. *Essential SNMP*. O'Reilly. 2.3.2

[131] David Meisner, Brian T. Gold, and Thomas F. Wenisch. Powernap: eliminating server idle power. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '09, pages 205–216, Washington, DC, USA, 2009. 1.3.3, 4.5, 4.6, 8.1.1

[132] 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*, ISCA '11, pages 319–330, San Jose, CA, USA, 2011. 2.3.1, 4.3, 4.5, 4.6

[133] David Meisner and Thomas F. Wenisch. Peak power modeling for data center servers with switched-mode power supplies. In *Proceedings of the 16th ACM/IEEE International Symposium on Low Power Electronics and Design*, ISLPED '10, pages 319–324, Austin, Texas, USA, 2010. 1.2.3

[134] David Meisner and Thomas F. Wenisch. DreamWeaver: architectural support for deep sleep. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '12, pages 313–324, London, England, UK, 2012. 8.2.1

[135] Frederic P. Miller, Agnes F. Vandome, and John McBrewster. *Advanced Configuration and Power Interface*. Alpha Press, 2009. 1.3.3

[136] Isi Mitrani. Managing performance and power consumption in a server farm. *Annals of Operations Research*, pages 1–14, 2012. 7.2.4

[137] Stephanie Mlot. Google Eyes Renewable Energy With N.C. Data Center Expansion. `http://www.pcmag.com/article2/0,2817,2417967,00.asp`, April 2013. 1.2.1

[138] David Mosberger and Tai Jin. httperf—A Tool for Measuring Web Server Performance. *ACM Sigmetrics: Performance Evaluation Review*, 26:31–37, 1998. 2.3.2, 6.2

[139] Dushyanth Narayanan, Austin Donnelly, and Antony Rowstron. Write off-loading: practical power management for enterprise storage. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, FAST '08, pages 253–267, San Jose, CA, USA, 2008. 2.2.4

[140] Ripal Nathuji, Canturk Isci, and Eugene Gorbatov. Exploiting Platform Heterogeneity for Power Efficient Data Centers. In *Proceedings of the 4th International Conference on Autonomic Computing*, ICAC '07, Jacksonville, FL, USA, 2007. 8.2.2, 8.2.3, 9.2

[141] Ripal Nathuji, Aman Kansal, and Alireza Ghaffarkhah. Q-clouds: Managing performance interference effects for QoS-aware clouds. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 237–250, Paris, France, 2010. 2.1.2, 2.2, 4.1, 4.7.4, 5.1, 5.1, 5.8, 8.3

[142] Sergiu Nedevschi, Lucian Popa, Gianluca Iannaccone, Sylvia Ratnasamy, and David Wetherall. Reducing network energy consumption via sleeping and rate-adaptation. In *Proceedings of the 5th USENIX Symposium on Networked Sys-*

tems Design and Implementation, NSDI '08, pages 323–336, San Francisco, CA, USA, 2008. 8.2.1, 8.2.3

[143] M. E. J. Newman. Power laws, Pareto distributions and Zipf's law. *Contemporary Physics*, 46:323–351, December 2005. 2.3.2, 3.3.1, 4.2.1, 5.2.1, 6.2

[144] NLANR. National Laboratory for Applied Network Research. Anonymized access logs. ftp://ftp.ircache.net/Traces/, 1995. 4.1(c), 4.1(d), 4.1(e), 4.3.5, 4.3.6, 4.5.1, 4.9, 4.10, 4.5.2, 4.5.3, 4.11, 4.12, 4.13, 4.6, 4.7.3, 4.4, 4.7.3, 5.4.2

[145] Kathleen Ohlson. Victorias secret knows ads, not the web. *Computer World*, February 1999. 1.3.3, 3.2.2, 5.1, 5.8

[146] Michael A. Olson, Keith Bostic, and Margo Seltzer. Berkeley DB. In *Proceedings of the 1999 USENIX Annual Technical Conference*, ATC '99, pages 183–191, Monterey, CA, USA, 1999. 2.3.2, 3.3.1, 6.2

[147] Peter Pachal. Amazon apologizes for cloud outage, issues credit to customers. *PCMag*, April 2011. 3.2.2, 5.1, 5.4.3, 5.8

[148] Pradeep Padala, Kai-Yuan Hou, Kang G. Shin, Xiaoyun Zhu, Mustafa Uysal, Zhikui Wang, Sharad Singhal, and Arif Merchant. Automated control of multiple virtualized resources. In *Proceedings of the 4th ACM European Conference on Computer Systems*, EuroSys '09, pages 13–26, Nuremberg, Germany, 2009. 5.1, 5.1

[149] Darshan S. Palasamudram, Ramesh K. Sitaraman, Bhuvan Urgaonkar, and Rahul Urgaonkar. Using batteries to reduce the power costs of internet-scale distributed networks. In *Proceedings of the 3rd ACM Symposium on Cloud Computing*, SoCC '12, pages 1–14, San Jose, CA, USA, 2012. 8.4

[150] Trevor Pering, Tom Burd, and Robert Brodersen. The simulation and evaluation of dynamic voltage scaling algorithms. In *Proceedings of the International Symposium on Low Power Electronics and Design*, ISLPED '98, pages 76–81, Monterey, CA, USA, 1998. 4.3.4

[151] Christian Poellabauer, Leo Singleton, and Karsten Schwan. Feedback-Based Dynamic Voltage and Frequency Scaling for Memory-Bound Real-Time Applications. In *Proceedings of the 11th IEEE Real Time on Embedded Technology and Applications Symposium*, RTAS '05, pages 234–243, San Francisco, CA, USA, 2005. 8.1.2

[152] W. Qin and Q. Wang. Modeling and control design for performance management of web servers via an IPV approach. *IEEE Transactions on Control Systems Technology*, 15(2):259–275, March 2007. 2.2, 4.1

[153] Ramya Raghavendra, Parthasarathy Ranganathan, Vanish Talwar, Zhikui Wang, and Xiaoyun Zhu. No "Power" Struggles: Coordinated Multi-Level Power Management for the Data Center. In *Proceedings of the 13th Interna-*

tional Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '08, pages 48–59, Seattle, WA, USA, 2008. 1, 8.4

[154] Luiz Ramos and Ricardo Bianchini. C-Oracle: Predictive Thermal Management for Data Centers. In *Proceedings of the 14th IEEE International Symposium on High-Performance Computer Architecture*, HPCA '08, pages 111–122, Salt Lake City, UT, USA, 2008. 8.4

[155] Parthasarathy Ranganathan, Phil Leech, David Irwin, David Irwin, and Jeffrey Chase. Ensemble-level Power Management for Dense Blade Servers. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture*, ISCA '06, pages 66–77, Boston, MA, USA, 2006. 8.4

[156] Joshua Reich, Michel Goraczko, Aman Kansal, and Jitendra Padhye. Sleepless in seattle no longer. In *Proceedings of the 2010 USENIX Annual Technical Conference*, ATC '10, pages 315–328, Boston, MA, USA, 2010. 8.2.1, 8.2.3

[157] Chuangang Ren, Di Wang, Bhuvan Urgaonkar, and Anand Sivasubramaniam. Carbon-Aware Energy Capacity Planning for Datacenters. In *Proceedings of the 20th IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, MASCOTS '12, pages 391–400, Arlington, VA, USA, 2012. 8.4

[158] Erik Riedel, Christos Faloutsos, Garth A. Gibson, and David Nagle. Active Disks for Large-Scale Data Processing. *IEEE Computer*, 34(6):68–74, June 2001. 8.2.3

[159] Alma Riska and Evgenia Smirni. M/G/1-type Markov processes: A tutorial. In *Performance Evaluation of Complex Systems: Techniques and Tools*, pages 36–63. Springer, 2002. 2

[160] Sheldon M. Ross. *Introduction to Probability Models*. Academic Press, Inc., Orlando, FL, USA, seventh edition, 2000. 3.1.1, 3.1.1, 3.1.1, 4.7.3, 7.1, 7.4

[161] Yasushi Saito, Svend Frølund, Alistair Veitch, Arif Merchant, and Susan Spence. FAB: Building Distributed Enterprise Disk Arrays from Commodity Components. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '04, pages 48–58, Boston, MA, USA, 2004. 2.2.4

[162] Bianca Schroeder, Eduardo Pinheiro, and Wolf-Dietrich Weber. DRAM errors in the wild: a large-scale field study. In *Proceedings of the 11th International Joint Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '09, pages 193–204, Seattle, WA, USA, 2009. 3.2.2, 5.1, 5.4.3, 5.8

[163] George Schussel. Client/Server: Past, Present and Future. `http://www.dciexpo.com/geos/dbsejava.htm`, September 2006. 5.1, 5.8

[164] Navin Sharma, Sean Barker, David Irwin, and Prashant Shenoy. Blink: Man-

aging server clusters on intermittent power. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '11, pages 185–198, Newport Beach, CA, USA, 2011. 6.3.1, 6.3.1, 6.3.3

[165] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Proceedings of the 26th IEEE Symposium on Mass Storage Systems and Technologies*, MSST '10, pages 1–10, Incline Village, NV, USA, 2010. 1.1

[166] Suresh Siddha, Venkatesh Pallipadi, and Arjan Van De Ven. Getting maximum mileage out of tickless. In *Proceedings of the 2007 Ottawa Linux Symposium*, OLS '07, pages 201 – 208, Ottawa, Ontario, Canada, 2007. 8.1.1

[167] Bill Snyder. Server virtualization has stalled, despite the hype. http://www.infoworld.com/print/146901, December 2010. 2.1.1, 4.1

[168] Etienne Le Sueur and Gernot Heiser. Slow down or sleep, that is the question. In *Proceedings of the 2011 USENIX Annual Technical Conference*, ATC '11, pages 217–222, Portland, OR, USA, 2011. 4.5

[169] Alexander S. Szalay, Gordon C. Bell, H. Howie Huang, Andreas Terzis, and Alainna White. Low-power amdahl-balanced blades for data intensive computing. *SIGOPS Operating Systems Review*, 44(1):71–75, 2010. 8.2.3

[170] Eno Thereska, Austin Donnelly, and Dushyanth Narayanan. Sierra: practical power-proportionality for data center storage. In *Proceedings of the 6th European Conference on Computer Systems*, EuroSys '11, pages 169–182, Salzburg, Austria, 2011. 2.1.2, 2.2.4, 5.8, 6.1, 8.1.1

[171] Naishuo Tian, Quan-Lin Li, and Jinhua Gao. Conditional stochastic decompositions in the M/M/c queue with server vacations. *Stochastic Models*, 15(2):367–377, 1999. 7.2.3

[172] Beth Trushkowsky, Peter Bodík, Armando Fox, Michael J. Franklin, Michael I. Jordan, and David A. Patterson. The SCADS director: scaling a distributed storage system under stringent performance requirements. In *Proceedings of the 9th USENIX conference on File and Stroage Technologies*, FAST '11, pages 163–176, San Jose, CA, USA, 2011. 5.1, 5.1, 5.7, 5.8

[173] Bhuvan Urgaonkar and Abhishek Chandra. Dynamic Provisioning of Multi-tier Internet Applications. In *Proceedings of the 2nd International Conference on Automatic Computing*, ICAC '05, pages 217–228, Seattle, WA, USA, 2005. 2.1.3, 2.2, 2.2.3, 2.3.1, 4.1, 4.3, 4.3.2, 5.1, 5.1, 5.8

[174] Bhuvan Urgaonkar, Giovanni Pacifici, Prashant Shenoy, Mike Spreitzer, and Asser Tantawi. An analytical model for multi-tier internet services and its applications. In *Proceedings of the 2005 ACM SIGMETRICS International*

Conference on Measurement and Modeling of Computer Systems, SIGMETRICS '05, pages 291–302, Banff, Alberta, Canada, 2005. 2.2, 4.1

[175] Bhuvan Urgaonkar and Prashant Shenoy. Cataclysm: Scalable overload policing for internet applications. *Journal of Network and Computer Applications*, 31(4):891 – 920, 2008. 5.7

[176] Bhuvan Urgaonkar, Prashant Shenoy, and Timothy Roscoe. Resource overbooking and application profiling in shared hosting platforms. *SIGOPS Operating Systems Review*, 36(SI):239–254, December 2002. 8.3

[177] B. Van Houdt and J.S.H. van Leeuwaarden. Triangular M/G/1-Type and Tree-Like Quasi-Birth-Death Markov Chains. *INFORMS Journal on Computing*, 23(1):165–171, 2011. 7.2.1

[178] JSH Van Leeuwaarden and EMM Winands. Quasi-birth-and-death processes with an explicit rate matrix. *Stochastic models*, 22(1):77–98, 2006. 7.2.1

[179] Nedeljko Vasić, Martin Barisits, Vincent Salzgeber, and Dejan Kostic. Making Cluster Applications Energy-Aware. In *Proceedings of the 1st Workshop on Automated Control for Datacenters and Clouds*, ACDC '09, pages 37–42, Barcelona, Spain, 2009. 2.2.4

[180] Vijay Vasudevan, David Andersen, Michael Kaminsky, Lawrence Tan, Jason Franklin, and Iulian Moraru. Energy-efficient cluster computing with FAWN: workloads and implications. In *Proceedings of the 1st International Conference on Energy-Efficient Computing and Networking*, e-Energy '10, pages 195–204, Passau, Germany, 2010. 8.2.3

[181] R.K. Venkatesan, S. Herr, and E. Rotenberg. Retention-aware placement in DRAM (RAPID): software methods for quasi-non-volatile DRAM. In *Proceedings of the 12th International Symposium on High-Performance Computer Architecture*, HPCA '06, pages 155–165, Austin, TX, USA, 2006. 8.1.1

[182] Akshat Verma, Puneet Ahuja, and Anindya Neogi. pMapper: power and migration cost aware application placement in virtualized systems. In *Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware*, Middleware '08, pages 243–264, Leuven, Belgium, 2008. 8.3

[183] Akshat Verma, Gargi Dasgupta, Tapan Kumar Nayak, Pradipta De, and Ravi Kothari. Server workload analysis for power minimization using consolidation. In *Proceedings of the 2009 USENIX Annual Technical Conference*, USENIX '09, pages 355–368, San Diego, CA, USA, 2009. 4.3.1, 4.3.4, 8.3

[184] Akshat Verma, Ricardo Koller, Luis Useche, and Raju Rangaswami. SRCMap: energy proportional storage using dynamic consolidation. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies*, FAST '10, pages 267–280, San Jose, CA, USA, 2010. 8.1.1

[185] Thiemo Voigt, Renu Tewari, Douglas Freimuth, and Ashish Mehra. Kernel Mechanisms for Service Differentiation in Overloaded Web Servers. In *Proceedings of the 2001 USENIX Annual Technical Conference*, ATC '01, pages 189–202, Boston, MA, USA, 2001. 5.7

[186] L. A. Wald and S. Schwarz. The 1999 southern california seismic network bulletin. *Seismological Research Letters*, 71:401–422, July 2000. 1.3.3, 3.2.2, 5.1

[187] C.A. Waldspurger, T. Hogg, B.A. Huberman, J.O. Kephart, and W.S. Stornetta. Spawn: A distributed computational economy. *IEEE Transactions on Software Engineering*, 18(2):103–117, February 1992. 2.2, 2.2.5

[188] Kai Wang, Minghong Lin, Florin Ciucu, Adam Wierman, and Chuang Lin. Characterizing the impact of the workload on the value of dynamic resizing in data centers. *Performance Evaluation Review*, 40(1):405–406, June 2012. 2.1.2

[189] Peijian Wang, Yong Qi, Xue Liu, Ying Chen, and Xiao Zhong. Power management in heterogeneous multi-tier web clusters. *International Conference on Parallel Processing*, pages 385–394, 2010. 2.1.2, 2.2, 2.2.2

[190] X. Wang and M. Chen. Cluster-level feedback power control for performance optimization. In *Proceeding of the 14th IEEE International Symposium on High-Performance Computer Architecture*, HPCA '08, pages 101–110, Salt Lake City, UT, USA, 2008. 2.1.2, 2.2, 4.1

[191] Xiaoying Wang, Zhihui Du, Yinong Chen, and Sanli Li. Virtualization-based autonomic resource management for multi-tier web applications in shared data center. *Journal of Systems and Software*, 81(9):1591 – 1608, 2008. 2.1.2, 2.2

[192] M. Ware, K. Rajamani, M. Floyd, B. Brock, J.C. Rubio, F. Rawson, and J.B. Carter. Architecting for power management: The IBM POWER7 approach. In *Proceedings of the 16th IEEE International Symposium on High Performance Computer Architecture*, HPCA '10, pages 1–11, Bangalore, India, 2010. 4.5

[193] Charles Weddle, Mathew Oldham, Jin Qian, An-I Andy Wang, Peter Reiher, and Geoff Kuenning. PARAID: a gear-shifting power-aware RAID. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies*, FAST '07, pages 245–260, San Jose, CA, USA, 2007. 8.1.1

[194] P.D. Welch. On a generalized $M/G/1$ queueing process in which the first customer of each busy period receives exceptional service. *Operations Research*, 12:736–752, 1964. 2.1.3, 3.1, 7.1, 7.5.3

[195] Daniel Wong and Murali Annavaram. KnightShift: Scaling the Energy Proportionality Wall through Server-Level Heterogeneity. In *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '12, pages 119–130, Vancouver, British Columbia, Canada, 2012. 8.2.2

[196] Timothy Wood, Prashant J. Shenoy, Arun Venkataramani, and Mazin S. Yousif. Black-box and gray-box strategies for virtual machine migration. In *Proceedings of the 4th USENIX Conference on Networked Systems Design and Implementation*, NSDI '07, pages 229–242, Cambridge, MA, USA, 2007. 2.1.2, 2.2, 4.1

[197] Xiuli Xu and Naishuo Tian. The M/M/c Queue with (e, d) Setup Time. *Journal of Systems Science and Complexity*, 21:446–455, 2008. 7.2.3

[198] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 265–278, Paris, France, 2010. 3.1.2

[199] Zhe George Zhang and Naishuo Tian. Analysis on queueing systems with synchronous vacations of partial servers. *Performance Evaluation*, 52(4):269–282, 2003. 7.2.3

[200] Wei Zheng, Ana P. Centeno, Frederic Chong, and Ricardo Bianchini. LogStore: Toward energy-proportional storage servers. In *Proceedings of the 2012 ACM/IEEE International Symposium on Low Power Electronics and Design*, ISLPED '12, pages 273–278, Redondo Beach, CA, USA, 2012. 8.1.1, 8.1.2

[201] Qingbo Zhu, Zhifeng Chen, Lin Tan, Yuanyuan Zhou, Kimberly Keeton, and John Wilkes. Hibernator: Helping disk arrays sleep through the winter. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, SOSP '05, pages 177–190, Brighton, United Kingdom, 2005. 8.1.2