Title
Enabling efficiency in data center systems

Permalink
https://escholarship.org/uc/item/72s9v3bt

Authors
McCullough, John Carleton

Publication Date
2012

Peer reviewed|Thesis/dissertation
UNIVERSITY OF CALIFORNIA, SAN DIEGO

Enabling Efficiency in Data Center Systems

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

in

Computer Science

by

John C. McCullough

Committee in charge:

Professor Alex C. Snoeren, Chair
Professor Tara Javidi
Professor Rajesh Gupta
Professor Stefan Savage
Professor Geoff Voelker

2012
The dissertation of John C. McCullough is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

________________________________________________________________________

________________________________________________________________________

________________________________________________________________________

________________________________________________________________________

________________________________________________________________________

Chair

University of California, San Diego

2012
DEDICATION

To my family, because you are the best.
I have never let my schooling interfere with my education.

—Mark Twain
# TABLE OF CONTENTS

Signature Page ............................................................... iii

Dedication ........................................................................ iv

Epigraph ........................................................................... v

Table of Contents ............................................................ vi

List of Figures .................................................................. ix

List of Tables .................................................................... xi

Acknowledgements ........................................................... xii

Vita ................................................................................. xv

Abstract of the Dissertation ............................................. xvi

Chapter 1  Introduction ...................................................... 1
  1.1 Modern data centers .................................................. 2
  1.2 Towards efficiency .................................................... 4
  1.3 Contributions .......................................................... 6
  1.4 Organization ........................................................... 8

Chapter 2  Background and Related Work .......................... 9
  2.1 Computational Efficiency .......................................... 10
  2.2 Storage efficiency .................................................... 12
  2.3 Networking efficiency .............................................. 13
  2.4 Energy efficiency ..................................................... 14
  2.5 Summary .................................................................. 15

Chapter 3  Scc ................................................................. 16
  3.1 Introduction ............................................................. 16
  3.2 Problem setting and overview .................................... 19
  3.3 Inputs to scc ............................................................ 21
     3.3.1 Specifying SLAs ............................................... 21
     3.3.2 Cluster building blocks ...................................... 23
     3.3.3 Characterizing applications ................................ 25
  3.4 Implementation of scc .............................................. 28
     3.4.1 Overview ........................................................ 28
     3.4.2 Resources for datasets ...................................... 30
     3.4.3 Task Resources ............................................... 32
3.4.4 Optimizing server costs .................................. 33
3.5 Evaluation .................................................. 33
  3.5.1 Methodology ............................................. 33
  3.5.2 Photo sharing ............................................ 35
  3.5.3 Product search .......................................... 38
  3.5.4 Sorting binary tuples ................................... 40
3.6 Discussion .................................................. 41
  3.6.1 Robustness of $scc$’s output .......................... 41
  3.6.2 Informing application development .................... 43
  3.6.3 Extensibility of $scc$ ................................. 44
3.7 Related work ................................................. 45
3.8 Summary ..................................................... 46

Chapter 4 Stout .................................................. 48
  4.1 Introduction ................................................ 49
  4.2 Background ............................................... 51
  4.3 Adaptive Batching ....................................... 53
    4.3.1 Overlapped Request Processing ..................... 54
    4.3.2 Updating the Batching Interval ...................... 56
  4.4 Implementation ......................................... 61
    4.4.1 Key Ownership ....................................... 62
    4.4.2 Stout Internal Architecture ......................... 63
    4.4.3 Persistence and Dependencies ....................... 63
    4.4.4 Stout API ............................................. 65
  4.5 Evaluation ............................................... 66
    4.5.1 Experimental Setup .................................. 67
    4.5.2 Batching and Write Collapsing ...................... 69
    4.5.3 Adaptive vs. Fixed Batching ......................... 71
    4.5.4 Dynamic Load Changes ............................... 74
    4.5.5 Fairness .............................................. 75
    4.5.6 Alternate Storage Layers ............................ 75
    4.5.7 Store Hiccups ....................................... 76
  4.6 Related work ............................................. 79
  4.7 Summary .................................................. 81

Chapter 5 Model-Based Power Characterization ................. 82
  5.1 Introduction ............................................. 83
  5.2 Related work ............................................. 86
  5.3 Power characterization ................................... 87
    5.3.1 Measurement granularity ............................ 88
    5.3.2 Accuracy ............................................ 89
    5.3.3 Overhead and complexity ............................ 90
  5.4 Power modeling ......................................... 91
LIST OF FIGURES

Figure 1.1: Data center overview. .................................................. 2
Figure 1.2: Amazon’s James Hamilton’s modeled monthly cost breakdown of
≈ $3.5M for a 50k-server data center running at 12MW total load
with a 3-year server replacement cycle and a 15-year data center
lifecycle [61]. ................................................................. 3
Figure 1.3: Data center usage apportioned by application/service. ............. 4
Figure 2.1: Three-tier Web architecture. .......................................... 10
Figure 3.1: Overview of scc. .......................................................... 20
Figure 3.2: Example specifications of (a) SLAs for PhotoShare, (b) hardware
resources, and (c) application behavior for a particular deployment
of ProductSearch. .......................................................... 22
Figure 3.3: Interaction between tasks and datasets in example application Photo-
share. Edges between tasks and datasets represent I/O with di-
rection differentiating input and output. Dotted edges indicate task
dependencies. ............................................................... 25
Figure 3.4: Summary of scc’s procedure for determining a cost-effective cluster
configuration that satisfies target SLAs, given a particular assign-
ment of storage types to datasets. .......................................... 29
Figure 3.5: Validation of cluster output by scc for particular SLA values in the
three application cases. ...................................................... 35
Figure 3.6: Cost versus SLA distribution output by scc for PhotoShare. Note
log scale on y axis. .......................................................... 36
Figure 3.7: Transition in scc’s output for ProductSearch from Config1 at 12
queries/minute to Config2 at 13 queries/minute. ......................... 39
Figure 4.1: Stout in a data center spreadsheet application. ....................... 52
Figure 4.2: An example of overlapped request processing. ...................... 54
Figure 4.3: The internal architecture of Stout. ................................... 62
Figure 4.4: Data structures for Stout’s dependency map. ....................... 62
Figure 4.5: An example use case of a spreadsheet application interacting with
Stout. ........................................................................... 64
Figure 4.6: Two fixed batching intervals (10 ms, 20 ms) on a workload with
low write collapsing (10k documents) or high write collapsing (100
documents). ................................................................ 69
Figure 4.7: Two fixed batching intervals (10 ms, 20 ms) on identical workloads
with and without batching. ..................................................... 70
Figure 4.8: Mean response latency for writes: Stout versus fixed batching inter-
vals over a wide variety of loads. ........................................... 71
Figure 4.9: Mean response latency for reads: Stout versus fixed batching intervals over a wide variety of loads. Note that the y-axis is $4 \times$ smaller than in Figure 4.8.

Figure 4.10: Illustration of Stout finding correct batching interval when nearby fixed batching interval would generate queuing.

Figure 4.11: Stout outperforms a fixed batching interval after the load either increases or decreases.

Figure 4.12: Mean response latency for writes using PacificA: Stout and fixed intervals over a variety of loads.

Figure 4.13: Intermittent hiccups in store processing yield brief spikes in latency as measured from the middle tier. These measurements were taken with a 2-ms fixed batching interval and 3k requests/second.

Figure 4.14: Stout recovering from a store hiccup while operating at 3k requests/second.

Figure 5.1: Power breakdown for sample workloads.

Figure 5.2: Test harness configuration.

Figure 5.3: Modeling accuracy for total system power (Single-Core). Mean relative error is 1–3% across workloads.

Figure 5.4: Modeling accuracy for CPU power (Single-Core). Mean relative error is 2-6% across workloads.

Figure 5.5: Modeling accuracy for memory power (Single-Core). Mean relative error is 4–8% across workloads.

Figure 5.6: Modeling accuracy for total system power (Multi-core). Mean relative error is 2–6% across workloads.

Figure 5.7: Modeling accuracy for CPU power (Multi-core). Mean relative error is 10–14% across workloads, but as high as 150% for some workloads on the right.

Figure 5.8: Prediction error with hard disk (left) and newer technology SSD (right).

Figure 5.9: Variability in power consumption across two identical Intel Core i5-540m CPUs.
Table 3.1: Example set of cluster building blocks input to \textit{scc}. Cost is price plus energy costs for 3 years. \textit{scc} takes read and write gap parameters as input rather than IOPS. 18
Table 3.2: Different regimes based on SLA requirements in the cost-effective architecture for PhotoShare. 37
Table 3.3: Determining robustness of \textit{scc}’s output with respect to its input: (a) robustness of cluster configuration with respect to input values for a sample set of attributes, and (b) the change in hardware costs to which \textit{scc}’s storage decision for each dataset is most sensitive. 42
Table 4.1: How a service’s maximum throughput can increase by exploiting batching. 56
Table 4.2: Parameters to make measurements and comparisons robust to jitter. 57
Table 4.3: Parameters for gain and boundary conditions. These parameters are analogous to those in CTCP, e.g., $intrvl_{max}$ corresponds to $RTO_{max}$. 61
Table 4.4: Client API. All calls are asynchronous. 63
Table 5.1: Power characterization for the calpella platform. The 500-W ATX PSU that we use dissipates 20–26W due to conversion inefficiency and is not shown. 96
ACKNOWLEDGEMENTS

Many people have helped me on this journey through grad school and I am grateful for all of them.

First, I would like to thank my advisor, Professor Alex C. Snoeren, for his ongoing dedication. He has always given me guidance, wisdom, and time, even in extraordinary situations.

I would also like to thank Professor Geoff Voelker for his guidance and for keeping the warmth and spirit of the department alive.

I would like to thank the rest of my committee — Professors Stefan Savage, Rajesh Gupta, and Tara Javidi — for generously giving their time and feedback.

Thanks to my family for their support and for believing in me, regardless of how intelligent I was behaving at the time.

Thanks to Monique Germone for putting up with me on deadlines.

There are many fine people in the department who made graduate school a great experience. Among them are Julie Conner, Kathy Krane, and Jennifer Folkestad, who keep things lively and have always cut through the red tape with poise and aplomb.

Thanks to all of my co-authors, in alphabetical order, Alec Wolman, Amin Vahdat, George Porter, Harsha Madhyastha, Jaideep Chandrashekar, John Dunagan, Justin Ma, Michael Vrable, Nabil Schear, Qing Zhang, Rishi Kapoor, Sathyanarayan Kuppuswamy, and Yuvraj Agarwal. It has been a pleasure.

Thanks to my co-authors in non-publication, Jay Chen and Barath Raghavan, for the many hours and comraderie that come with the harsh realities of research.

Thanks to Barathan Balaji for his help on hardware.

Special thanks to John Dunagan for his continuing guidance and mentorship.

Thanks to my many friends and colleagues. Thanks to Aaron Arvey for being a far away friend. Thanks to Alex Rasmussen for boomslanging it away. Thanks to Alvin Auyoung for his calm wisdom. Thanks to Andreas Pitsillidis for promoting Windows Phone. Thanks to Bhanu Vattikonda for begin a good guy. Thanks to Brian Kantor for building fun things. Thanks to Brian McFee for being more like that. Thanks to Catherine Wah for chilling like a villain. Thanks to Chip Killian for covering the endzone. Thanks to Chris Edwards for presenting in vim. Thanks to Chris Fleizach for a
cultural exchange. Thanks to Chris Kanich for the get lunch crew. Thanks to Christos Kozanitis for the persistent hello. Thanks to Cindy Moore for keeping the systems alive. Thanks to Cynthia Taylor for being a steadfast lunch buddy. Thanks to Danny Huang for being second Danny. Thanks to Danny Turner for turning on the lights. Thanks to Ding Yuan for his friendly encouragement. Thanks to Dionysios Logothetis for his friendly perspective. Thanks to Diwaker Gupta for setting the bar high. Thanks to Don Jang for his cultural fluency. Thanks to Feng Lu for being so friendly. Thanks to Gjergji Zyba for missing all of the flights. Thanks to Grant Jordan for taking up slack. Thanks to Harsha Madhyastha for graciously sharing. Thanks to James Anderson for the occasional quiet contemplation. Thanks to Jeanne Albrecht for being Lab Mom. Thanks to Kashi Vishwanath for rocking on. Thanks to Keaton Mowrey for being tall. Thanks to Kevin Webb for being a champ. Thanks to Krista Davis and Mike Stepp for pulling me toward the borg. Thanks to Lonnie Liu for following me through the Portuguese rain. Thanks to Marti Motoyama for his furtive pacing. Thanks to Marvin McNett for keeping the cluster alive. Thanks to Matthew Der for his cheer. Thanks to Meg Walraed-Sullivan for shifting allegiance to Monique. Thanks to Michael Vrable for doing everything. Thanks to Mike Conley for hearting house music. Thanks to Ming Kawaguchi for his high opinions. Thanks to Natalie Castellana for the long walks to the bio-vats. Thanks to Nathan Goulding for breaking down the foggy mountain. Thanks to Neha Chachra for smiling at everyone as they walk into lab. Thanks to Nima Nikzad for keeping it classy. Thanks to Panagiotis Vekris for chasing locks. Thanks to Patrick Verkaik for being old. Thanks to Pat Rondon for the rain, the rain. Thanks to Peng Huang for being reliable. Thanks to Priya Mahadevan for all the advice. Thanks to Radhika Niranjan Mysore for begin a good sport. Thanks to Ramana Kompella for keeping it real. Thanks to Ravi Chugh for a refined swing. Thanks to Rishi Kapoor for grinding the experiments. Thanks to Ross Tate for his uncategorical cheer. Thanks to Ryan Braud for the trophies and the excitement. Thanks to Shaan Mahnunani for endless enthusiasm. Thanks to Tristan Halvorson for picking up the helm. Thanks to Xiao Ma for standing up for what’s right. Thanks to Zach Tatlock for being so smiley.

Thanks to everyone else, whose names may not have come to mind, this sentence, right here, is just for you.
Chapter 3, in part, is a reprint of material as it appears in the article “scc: Cluster Storage Provisioning Informed By Application Characteristics and SLAs” by Harsha Madhyastha, John C. McCullough, George Porter, Rishi Kapoor, Stefan Savage, Alex C. Snoeren, and Amin Vahdat which appears in Proceedings of the USENIX Conference on File and Storage Technologies, 2012. The dissertation author was a primary investigator and co-author of this article.

Chapter 4, in part, is a reprint of material as it appears in the article “Stout: An Adaptive Interface to Scalable Cloud Storage” by John C. McCullough, John Dunagan, Alec Wolman, and Alex C. Snoeren which appears in Proceedings of the USENIX Annual Technical Conference, 2010. The dissertation author was the primary investigator and author of this article.

Chapter 5, in part, is a reprint of material as it appears in the article “Evaluating the Effectiveness of Model-Based Power Characterization” by John C. McCullough, Yuvraj Agarwal, Jaideep Chandrashekar, Sathyanarayan Kuppuswamy, Alex C. Snoeren, and Rajesh K. Gupta which appears in Proceedings of the USENIX Annual Technical Conference, 2011. The dissertation author was the primary investigator and author of this article.
VITA

2006 Bachelor of Science in Computer Science
Harvey Mudd College

2008 Master of Science in Computer Science
University of California, San Diego

2012 Doctor of Philosophy in Computer Science
University of California, San Diego

PUBLICATIONS


ABSTRACT OF THE DISSERTATION

Enabling Efficiency in Data Center Systems

by

John C. McCullough

Doctor of Philosophy in Computer Science

University of California, San Diego, 2012

Professor Alex C. Snoeren, Chair

Internet-scale computer systems have established themselves as the dominant platform for the majority of popular computing today. These systems handle workloads that span Web processing to scalable storage to batch processing. The massive quantity of servers that handle these workloads are housed in data centers and require large quantities of power and cooling. To extract the most benefit from these resources, we must ensure that they are utilized efficiently.

In this dissertation, we address three important challenges in efficient data-center computing. First, we present the storage configuration compiler that combines an application model and workload characterization to identify low-cost hardware configurations. Next, we present Stout, a software layer that enables more efficient use of shared storage. Finally, we consider the accuracy and limitations of several lightweight
power modeling techniques and the potential impact on dynamic power management. Together, they can significantly lower hardware costs and ensure effective consideration of energy management techniques.
Chapter 1

Introduction

The scale of modern computation is making efficiency paramount. The Internet contains more than 200 exabytes of data [64]. Crawling and indexing all of it is a substantial undertaking and quickly searching for something means that a search query may touch thousands of computers. Similarly, contemporary e-mail processing can require substantial computing resources to provide a pleasant experience. Each message must pass through various stages of unsolicited advertising filtering, anti-virus and security checks and all of these stages necessitate a large cluster to operate at scale. For companies like Google, Amazon, Microsoft, and Facebook to provide their various services to millions of users, they need warehouses full of computers. Each of these warehouse-sized data centers can contain on the order of fifty thousand to several hundred thousand machines. A recent Google data center was designed to hold 45,000 and Microsoft’s Chicago data center is designed to hold 300,000 [92]. Many large technology companies have server deployments of one hundred thousand to nine hundred thousand total machines [92]. The scale of these data centers implies that any inefficiency at the level of a single computer is amplified across the machines in a single data center, and even across all of the data centers a company has. Consequently, it is important to invest time and effort into maximizing the efficiency of the software, the efficiency of individual machines, and the efficiency of the data center infrastructure as a whole.
1.1 Modern data centers

A data center houses a large number of servers that are connected to each other and the outside world. To accomplish this goal, a data center must provide power, cooling, and network connectivity (Figure 1.1). The servers are typically arranged in racks, and the racks are aligned in rows. Power arrives in high voltage form and must be transformed and distributed to the servers. Backup power generators and batteries are common. The servers must be cooled, first by filtering outside air of particulate matter, then chilling it as necessary using large scale air conditioning components, next using fans to move the air to the servers, and finally either recycling the hot air or venting it to the outside. The capacity and cost of the cooling and power equipment are dictated by the requirements of the servers. The mechanisms for all of these are evolving with each new data center to provide more efficient computing resources.

![Data center overview](image)

**Figure 1.1:** Data center overview.

As shown in the monthly cost breakdown in Figure 1.2, we see that the principle cost comes from the servers themselves followed by the power and power distribution hardware to support them. The relative costs of individual servers and the technology used to power and cool them in a given data center may differ from what is used in this model, but it would take a drastic change to alter the overall picture. Given this state of affairs, the best way to maintain overall efficiency is to focus on the performance of the services running on the servers and to ensure that the power is put to good use.
Figure 1.2: Amazon’s James Hamilton’s modeled monthly cost breakdown of $\approx 3.5M\text{ }$ for a 50k-server data center running at 12MW total load with a 3-year server replacement cycle and a 15-year data center lifecycle [61].

To understand the efficiency of the data center we need to understand the efficiency of the individual applications and services. The thousands of servers in a data center are apportioned to different applications and services. In some situations, an application operates on an isolated set of servers, either for performance or management reasons. That set of servers is then broken down into individual tasks or groups of tasks. A three-tiered Web-application is a common example. It incorporates a front-end Web interface tier, an application logic tier, and a storage tier (Figure 1.3a). The major drawback of isolated applications comes in over-provisioning [28, 43]. Applications are provisioned to handle the everyday user load as well as the occasional surge in traffic demand. When the load variation for multiple applications is independent the server resources often sit idle. Sharing storage (Figure 1.3) and other resources allows different applications to take advantage of multiplexing to reduce over-provisioning and other costs.

This sharing of resources also reduces programming effort and operational costs.
Programming a distributed storage system that can scale across thousands of servers requires specialized knowledge. The storage software must provide performance and reliability in the face of hardware and network failures. Furthermore, different types of storage necessitate different architectures. Large binary objects, such as photos or movies, are likely to only be stored on hard disks or flash storage[15]. Small objects placed in key-value storage may be principally cached in memory and use hard disks strictly for recovery [44, 102]. Sharing the common services allows a smaller number of operational engineers and programmers to focus on that particular problem rather than spreading the attention and concerns across each individual application.

1.2 Towards efficiency

A server is put to the best use if the hardware and software are balanced. A “balanced system” [111] is one where the workload effectively bottlenecks the system on all available resources and no capacity goes to waste. This principle can be hard to achieve in practice due to the discrete nature of available resources, but it can be approached by optimizing software and identifying appropriate hardware.

The interactions of software and data influence the performance and behavior
of that software on a given piece of hardware. Optimizing and improving the software can make the system more efficient, but a hardware configuration may not be the best match to a given application. For instance, it is wasteful to add processors to a server configuration that is performance-bound by the storage subsystem. All of the different processors, memories, interconnect technologies, and storage mechanisms have their own trade-offs. The nuances of storage behavior are especially interesting because storage is a frequent system bottleneck.

Today, application providers can choose from a range of storage choices to provision the infrastructure for cluster-based applications. Storage technologies as diverse as DRAM, solid state drives (SSDs), and hard disks present complex trade-offs in cost, capacity, performance, and power consumption. Planning for the deployment of an application can be very challenging. Someone must combine the knowledge they have about the application and its behavior with the current workload. This task typically involves several rules of thumb that result in some standard Web server and some standard storage server, etc. Even if the configuration works at the current scale, it probably will not be balanced, and the inefficiencies of the setup will be amplified as the workload increases. Changes in the workload that lead to changes in overall behavior exacerbate the situation when solutions that consider the diversity of storage options are ignored.

An application deployment may have control over the balance of its own systems, but it may have little control over shared resources. Many cloud service providers use shared key-value storage. Sharing the storage layer means that the provisioning for the system is spread more evenly across multiple applications, but it can result in poor storage performance when other applications use it heavily. Poor storage performance degrades the overall application performance and it is important to enhance storage performance when possible. Applications do have opportunities to optimize their storage use by waiting longer and aggregating requests. Choosing the right waiting period is a trade-off between throughput and latency. However, always waiting longer can improve overall system throughput, but the wait degrades client-perceived latency when the storage system offers low latency service.

Another challenge in modern data centers is that systems are often under-utilized. This situation can be problematic because many computer servers have a large compo-
nent of base power when idle and some amount of dynamic power consumption that increases when the system is more heavily utilized. Barroso and Hölzle observed that in Google’s data centers, most of the machines are around 30% utilized most of the time [14]. The base power of the machine at idle means that the machines consume significant power even when they are not accomplishing anything. This observation led Barroso and Hölzle to call for power-proportional computing where the amount of power consumed is in proportion to the amount of work done.

Designing a system that is power proportional, either in terms of individual hardware or larger scale optimization, requires an understanding of the relationship between the workload and the hardware for anticipated range of utilization. There are ongoing efforts to perform cluster-wide power optimization [84, 127] as well as efforts to improve the efficiency and performance of server hardware [10, 60]. The hardware-level improvements, including multi-core and fine-grained power gating, make the relationship between the power draw and activity level increasingly complex. Optimizing the energy efficiency of a cluster requires insight into this relationship for cluster level dynamic power management, guiding improvements in hardware, and identifying appropriate software improvements. The overarching challenge comes in knowing whether the optimization techniques have an appreciable impact on the data center efficiency as a whole.

1.3 Contributions

In this dissertation, we address several challenges in efficient data-center computing. We first consider customizing the provisioned hardware to the actual applications to reduce cost and to improve resource usage. Because scalable key-value storage is a critical component in many applications, we developed Stout, a software layer that optimizes storage performance. Finally, because power is a significant fraction of costs, we consider the power consumption of modern hardware and its correspondence with application workload.

We developed techniques to model high-level application behavior and storage and system behavior to find low-cost hardware configurations that meet workload objec-
tives more effectively than today’s ad-hoc rules-of-thumb. This tool, the storage configuration compiler (scc), allows application designers to explore current and future workloads to make intelligent decisions in provisioning an efficient cluster. We evaluate scc against a range of application workloads and storage options to show that scc captures sufficient detail to prescribe the right combination of storage and server hardware at the right scale; modifying the architecture or reducing the scale leads to significant performance degradation. We found that scc often predicts heterogeneous architectures to meet application demands.

Next, we introduce Stout, a system that helps these applications adapt to variation in storage layer performance by treating scalable key-value storage as a shared resource requiring congestion control. Under light workloads, applications using Stout send requests to the store immediately, minimizing delay. Under heavy workloads, Stout automatically batches the application’s requests together before sending them to the store, resulting in higher throughput and reduced queuing delay. Stout treats store access as a congestion control problem, measuring application perceived latency and throughput of the store and dynamically adjusting the application’s grouping of requests to the store. To evaluate this algorithm, we implemented the Stout system and modified a real-world cloud service to use Stout. We found that in the presence of significant workload variation, Stout dramatically outperforms existing non-adaptive approaches with $34 \times$ lower latency compared to a throughput-oriented system and up to $3 \times$ as many requests compared to a latency-oriented system.

Finally, we evaluate whether power models developed for a time when system power consumption was more straightforward still apply to modern hardware and whether other modern techniques improve modeling accuracy. The models we consider are able to predict total system power to within 6% mean relative error for both single-core and multi-core scenarios, particularly when the base power of the system is high. However, for predicting subsystem power, we show that linear regression based models often perform poorly, and more complex non-linear models and support vector machines do only marginally better with 14% mean relative error and up to 150% worst case error. The poor subsystem power modeling is due to increased system and device complexity, and hidden power states that are not exposed to the operating system.
Furthermore, device variability complicates matters. Looking forward, while modeling techniques may suffice for some dynamic power management applications, our results motivate the need for pervasive, low-cost ways of measuring instantaneous subsystem power in commodity hardware.

As a result, we can improve efficiency by lowering hardware costs and understanding the impact of power optimization techniques on energy efficiency. We lower hardware costs with $scc$ by ensuring that we provision only hardware resources that the applications can use. We lower hardware costs with Stout by enabling more effective resource multiplexing. We scope the understanding of power adaptation techniques by evaluating modern power modeling and identifying challenges and limitations.

### 1.4 Organization

The remainder of this dissertation is organized as follows. Chapter 2 includes background material on work related to this dissertation. Chapter 3 describes the process for matching storage hardware to applications. Chapter 4 explores the design and implementation of Stout. Chapter 5 evaluates the performance of power models against modern computer hardware. Finally, Chapter 6 summarizes my work and discusses future work.
Chapter 2

Background and Related Work

The evolution of the Internet has yielded a fascinating array of services and technologies that help us locate information, communicate, and accomplish tasks on a scale not previously possible. The growth in required information storage and computation necessitates the hundreds of thousands of servers, as discussed in Chapter 1. The substantial resource requirements and costs encourage us to consider the resource efficiency of the diverse application workloads deployed on these computer clusters.

These large clusters are harnessed using some combination of the two dominant computational models, high-throughput batch processing and low-latency Web applications. The high-throughput batch processing workloads are typically deployed on MapReduce [38] or Dryad [67] style data processing systems. These are, in turn, backed by large-scale distributed storage systems similar to the Google File System [51] or BigTable [27]. Low-latency Web applications are frequently structured using the three tier architecture (Figure 2.1). The three tier architecture is comprised of a Web front-end, an application tier, and a storage tier. Storage tiers are traditionally comprised of SQL database servers. These database servers are frequently augmented with an in-memory key-value cache such as Memcached [44]. Others have started to employ persistent key-value stores such as Amazon’s Dynamo [39], RamCloud [102], or LinkedIn’s Project Voldemort [81].

As discussed earlier, these clusters are typically provisioned with a largely homogeneous set of machines. Many have observed that this state of affairs leads to different forms of inefficiency. One of the largest departures from the tradition of using
Figure 2.1: Three-tier Web architecture.

Similarly provisioned systems is the proposal of using low power processors to execute storage-bound workload. Others have identified an opportunity to improve performance and efficiency through diversity in storage technology.

2.1 Computational Efficiency

Different processors have different capabilities and consume varying amounts of power. Considered broadly, overall capabilities and power consumption increase when moving from embedded processors to mobile processors, to desktop processors, or to server processors. A system will be more efficient if we choose the processor with the capabilities that most closely match the balance of the workload and the other components in the system. As a reaction to the gross imbalances in many modern systems, there have been several projects exploring the applicability of lower-power and embedded processors to different classes of workloads.

Key-value storage systems spawned the initial work in the lower-power processing space. Key-value stores are frequently a thin software interface atop a storage system, and a server processor can support far more key-value requests than the storage medium or network interface can handle. The FAWN project took this situation to an extreme by pairing embedded processors with small amounts of flash storage to build a key-value store [10]. This platform necessitated a refined hash lookup scheme to fit the lookup table in the limited resources of the embedded platforms. But, in the end,
they surpassed the operations per joule of a desktop system by a factor of 5. Other data-intensive workloads perform such lightweight computations on the data that they also fall into the broad gap between storage speed and traditional processors. The Gordon project [24] paired Intel Atom processors with flash storage and achieved up to $2.5 \times$ better energy efficiency than a selection of Intel Core2Duos for data-heavy MapReduce workloads.

These large energy efficiency gains using low-power processors led others to consider whether low-power processors could be applied to alternative workloads. They found that the low-power processors fall short for different reasons on different workloads.

In some instances the lower-powered processors are inappropriate because of external resource limitations. In FAWNSort [129] the authors used a low-frequency Intel Core2 processor rather than the Atom because a one-pass in-memory sort of 10GB required more memory than an Atom processor supports.

In other instances the deficit arises from the components missing from lower-power processors that make them lower power in the first place. Janapa Reddi et al. performed an in-depth efficiency exploration comparing Intel Xeon processors to Intel Atoms on the Microsoft Search workload [112]. Microsoft Search involves an intense machine-learning kernel and the Atom processors are deficient in terms of branch prediction, cache size, and floating point capabilities. Even so, the Atom processors achieve high efficiency when considering the processor in isolation. But, when considering the motherboard and peripherals the Xeon processor achieves the highest efficiency. The FAWN authors observed shortcomings in cache sizes and memory throughput [128] and others have run into trouble when the workload exceeds simple string comparison. Keys et al. evaluated Atom, mobile, and server platforms for energy-efficiency in the data center [73]. They evaluated each platform using a selection of processor-heavy benchmarks and four Dryad benchmarks. They found that a power-optimized platform with an Intel Core2Duo mobile processor was the most energy efficient [73].

Another challenge with using lower-power CPUs is that you need many more individual computers to achieve the same throughput. This numerical increase leads to increased management overhead and potential degradation in performance. For in-
stance, Lang et al. observed that not all workloads are embarrassingly parallel. That is, a linear increase in machines may not imply a linear increase in throughput. They study low-power processors in the context of parallel databases and found that the power savings of the lower-power processors quickly diminishes in the presence of non-linear parallel speedup [77].

The gap in the performance between storage and processors is what allows FAWN-type systems to be energy efficient, but it is applicable even at higher levels of the memory hierarchy. Le Sueur et al. found that the power savings technique of dynamic voltage and frequency scaling only leads to efficiency gains for benchmarks that are memory bound rather than CPU bound [121]. In general, we can improve computational energy efficiency for the processor if it is not the bottleneck or if we can somehow change the application to execute more efficiently as the FAWN authors did with their key-value store.

We will likely continue to see improvements in the efficiency and performance of processors. While the most efficient processor choice for an application or set of applications may evolve, one of the challenges that we address in this dissertation is developing a methodology to identify the right processing, storage, and networking components for those applications.

### 2.2 Storage efficiency

Storage efficiency and performance present a unique challenge because achieved throughput can vary by several orders of magnitude depending on the workload. For hard disks, this comes from the construction of the device. A hard disk is a spinning platter with a read/write head. Data that is arranged in sequence can be read at high speed, while data that is randomly spread across the disk suffers from seek latency. Flash based storage is much faster for random reads and writes than hard disks, but there is still a large gap between sequential and random performance. Finding the balanced solution for a data center application involves choosing the right storage technique from the spectrum of price, performance, capacity, and reliability.

Beyond the physical media, there are variety of techniques to store the data to en-
hance reliability and performance. Ursa Minor observed that different datasets warrant different storage techniques to achieve higher performance [1]. In many situations, cluster storage is arranged in a RAID [104] system and the cluster manager chooses a RAID level that matches their expected need for drive failure, performance, and capacity. Different datasets and workloads have different storage demands, but the homogeneous configuration fixes all of the workloads to the same performance/reliability/encoding trade-off. These different workloads are better served by separate techniques. Large sequential datasets are best served by erasure coding typical of RAID systems and random access datasets can be better served by replicating data to increase the random-access capacity. Ursa Minor leveraged this idea and built a system for customized storage and encoding techniques and saw a $2-3 \times$ performance increase given the data access patterns for the different datasets.

The Ursa Minor approach works well if the data access patterns are known ahead of time. Hippodrome [11] follows a similar approach, but it also employs online algorithms to identify the needs of different datasets and enforce quality of service levels. While traditional quality of service methods do not apply to storage due to the inability to drop load and the peculiarities of the storage media, they are able to attain strong results using an iterative refinement of the deployed plans. Online refinement only considers optimizing over an existing set of resources and does not consider adding or changing hardware.

These approaches are very successful within the scope of measurable data streams for disk based systems. We address the challenges of improving application behavior to lower storage requirements as well as understanding the storage requirements as a function of the application workload rather than the storage stream itself.

### 2.3 Networking efficiency

The large datasets required for Internet scale computing will not fit on a single machine. Computations across datasets of this size require communication between servers. The network can bottleneck the performance of both the storage and computational components. This bottleneck comes not only from the speed of the network, but
also the nature of the topology. Many modern network hierarchies have strong connectivity near the servers, but when crossing the core to reach other servers the high over subscription factors limit the available bandwidth. Al-Fares, Loukissas, and Vahdat [7] employ a fat-tree topology to improve bisection bandwidth using low-cost networking hardware. Others have employed alternative network structures [57] and novel addressing mechanisms [56, 74, 95, 131]. These improvements in networking performance begin to provide opportunities for a better balance for throughput-intensive applications.

2.4 Energy efficiency

As discussed in Chapter 1, power proportionality and energy efficiency are important because many data center systems are not fully utilized. Over time there have been many efforts towards dynamic power management that involve turning off components to minimize power or optimizing lower-power states.

Some of the early work in this space focused on concentrating active load on a minimal set of servers and putting the inactive servers in low power states. Chase et al. [28] combine performance metrics and resource bids to allocate cluster resources to different applications. Using this technique they are able to reduce energy usage by 29%. Others employ the same idea in the context of moving virtual machines [103].

Others focus on fine-tuning the low-power states of the hardware within the server to achieve improved energy efficiency. Elnozahy et al. consider the use of dynamic voltage scaling and request batching to conserve energy while maintaining a specified quality of service level [42]. Raghavendra et al. devise a multi-level feedback controller to jointly optimize cooling efficiency, power-capping, and QoS metrics [110]. Tolia et al. consider optimizing cluster power using virtual machines, dynamic voltage and frequency scaling, and fan control to approach an energy-proportional system [127].

The aggregate power consumption of the server systems has a direct impact on the amount of heat generated by each server. If only a few servers are running at full utilization in one part of the data center, they require more cooling energy than if the servers are spread out more thinly. Ahmad and Vijaykumar consider jointly optimizing power-down policies with policies for cooling efficiency to achieve good energy
utilization while minimizing hotspots for reduced cooling energy usage [6].

In some storage environments, the amount of data that is frequently accessed is much smaller than the amount of data that is left idle. Pinheiro and Bianchini took advantage of this data distribution by consolidating frequently accessed data to a subset of the disks so that the others can be transitioned to low-powered modes. They found that single-speed disks are only able to conserve energy at very low loads because of the long transition time between the off and on states, but that two-speed disks are more effective [108]. Ganesh et al. apply the idea of log-structured file systems to disk storage and argue that it provides a natural power-saving opportunity for large-scale storage systems because writes are localized [49].

A challenge that remains in many of these approaches is ensuring that they actually save power. Le Sueur et al. [121] observed that the power savings available in modern processors from dynamic voltage and frequency scaling are greatly reduced because the lower base voltages leave little room for improvement. They also found that the lower-power states frequently result in reduced energy efficiency because the static power is not eliminated during the period of reduced execution speed. Meisner et al. [84] observed that most servers are not idle enough of the time to transition to existing low-power states and that fine-grained sleep transitions are necessary to achieve better energy efficiency.

Thus, before employing any of these dynamic power management techniques, it is critical to understand the trade-offs in energy consumption and the impact of changing hardware before introducing the associated performance overheads.

2.5 Summary

This chapter presents the nature of the state of the art in terms of server-level and data-center-level efficiency. In many cases, the solutions represent point solutions for system performance and the challenge is to find an integrated approach. In other cases, new storage techniques and requirements present new opportunities for improvement. Finally, the evolution of modern hardware necessitates that we reconsider existing assumptions and ascertain the value of an approach.
Chapter 3

Scc

As we established in Chapter 1, the traditional techniques and rules-of-thumb for scaling applications can lead to an unbalanced and expensive configuration. Storage is particularly challenging because the physical properties of storage can lead to very different performance when exposed to different access patterns. A better understanding of the application, the application’s workload, and the properties of the hardware can lead to a better match between application and hardware and, consequently, lower cost and more efficient hardware configurations.

In this chapter, we describe scc, a tool that takes a high-level application model, workload parametrization, and candidate hardware components and outputs low-cost candidate hardware configurations. This tool can assist application deployers in evaluating good hardware configurations spanning current and future workloads. First, we explore the challenges of identifying an appropriate high-level representation that flexibly represents storage and compute resources. Next, we describe how an application developer can translate their application into such a model, and then we demonstrate the tool’s effectiveness on a selection of representative applications deployed on a physical cluster.

3.1 Introduction

Today, application providers can choose from a range of storage choices to provision the infrastructure for cluster-based applications. Storage technologies as diverse
as DRAM, solid state drives (SSDs), and hard disks present complex trade-offs in cost, capacity, performance (along multiple dimensions), and power consumption. New storage technologies such as phase change memory [33] will soon further complicate the space.

 Provisioning, however, is based largely on rules of thumb and best practices. Applications are broadly categorized as storage, compute, or memory intensive and are typically deployed on homogeneous clusters heavy on the corresponding resource. As application load increases, deployments are “scaled out” by simply adding more storage and compute in the same configuration. Not only does this state of affairs fail to take full advantage of the diversity of available storage choices, but the increasing scale of deployments makes such inefficiencies worse; inefficiencies multiplied over thousands of servers can have substantial costs. In the scale-out model, a poor initial choice can greatly inflate expenses.

 In this chapter, we pursue an alternate approach—the automated selection of cluster storage configurations based on formal specifications of applications, hardware, and workloads. Initially, such an approach places significant burden on those developing and deploying applications to characterize applications and workloads. However, the resultant savings in cost necessary to satisfy Service Level Agreements (SLAs) can be substantial.

 Our primary contributions in implementing this approach are two-fold. First, we determine how the characteristics of applications, workloads, and hardware should be specified in order to automate the selection of cluster configurations. To do so, we study several representative deployment scenarios and identify a parsimonious yet sufficiently expressive set of parameters that capture the trade-offs offered by different types of storage devices and the varying demands across application components. Though others have pursued a similar approach of formally specifying workloads and hardware [8, 11, 137], we extend this approach to account for various types of storage media (e.g., disk, SSD, and DRAM) and to jointly capture storage and compute requirements of applications. We show that it is feasible to concisely summarize the most salient parameters that determine the resource requirements of specific application deployments, thus minimizing the burden of formal specification.
Table 3.1: Example set of cluster building blocks input to scc. Cost is price plus energy costs for 3 years. scc takes read and write gap parameters as input rather than IOPS.

<table>
<thead>
<tr>
<th>Resource</th>
<th>MB/s</th>
<th>IOPS</th>
<th>Watts</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.2K Disk (500 GB)</td>
<td>90 (R)</td>
<td>125 (R)</td>
<td>5</td>
<td>$213</td>
</tr>
<tr>
<td></td>
<td>90 (W)</td>
<td>125 (W)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15K Disk (146 GB)</td>
<td>150 (R)</td>
<td>285 (R)</td>
<td>2.3</td>
<td>$296</td>
</tr>
<tr>
<td></td>
<td>150 (W)</td>
<td>285 (W)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SSD (32 GB)</td>
<td>250 (R)</td>
<td>2500 (R)</td>
<td>2.4</td>
<td>$456</td>
</tr>
<tr>
<td></td>
<td>80 (W)</td>
<td>1000 (W)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DRAM (1 GB)</td>
<td>12.8K (R)</td>
<td>1.6B (R)</td>
<td>3.5</td>
<td>$35</td>
</tr>
<tr>
<td></td>
<td>12.8K (W)</td>
<td>1.6B (W)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU core</td>
<td>-</td>
<td>-</td>
<td>20</td>
<td>$137</td>
</tr>
</tbody>
</table>

Second, we develop scc, a storage configuration compiler that takes specifications of applications, workloads, and hardware as input, automates the navigation of the large space of storage configurations, and zeroes in on the configuration that meets application SLAs at minimum cost. To evaluate scc, we experiment with three distributed applications with distinctly different workload characteristics: 1) ProductSearch, a product search webservice modeled on Google Merchant Center [54], 2) Terasort, a MapReduce job to sort large tuple collections, and 3) PhotoShare, a photo-sharing Web service modeled on Flickr. By deploying these applications on a range of cluster configurations and measuring application performance on these configurations, we present empirical evidence that scc is expressive enough to capture the needs of a range of applications.

In developing scc and applying it to diverse application workloads, we make three key observations. First, the right choice of storage configuration depends not only on the storage capacity and I/O needs of the application, but also on the application’s compute requirements and on the types of server configurations available. When an application performs a set of operations in sequence, the resources assigned to serve
each of these operations must be jointly optimized to satisfy the performance bound on the sequence of operations at minimum cost. For example, in an application that performs an I/O operation on some data followed by some computation, the storage type assigned to the data depends on the amount of computation. When the computation consumes significant time, the data may need to be stored on fast storage like SSDs to meet performance bounds, whereas when compute time is low, there is greater slack in performing the I/O and hence, slower cheaper storage like disk may suffice.

Second, we find that clusters with heterogeneity—rather than the conventional homogeneity—across servers are necessary to optimize cost. The resources required differ across application components because of varying ratios of capacity, compute, and I/O throughput needs across components. For example, in a deployment of the photo-sharing Web service, it may be cheaper to store photos on disk and cache thumbnails in DRAM; storing both on disk or both in DRAM may result in higher cost due to higher I/O throughput needs from thumbnails or higher storage capacity needs of photos, respectively. As a result, scc’s suggested configuration meets performance SLAs at low cost. For example, in experiments with Terasort, we find that scc meets performance requirements at 15–20% lower cost than a homogeneous configuration recommended based on best practices.

Finally, we also find that the most cost-effective cluster architecture depends not only on the application being provisioned but also on the workload and performance requirements. Data that was initially capacity-bound may become I/O-bound at higher loads, calling for shifts from high capacity but slow storage, e.g., disks, to low capacity but fast storage, e.g., SSDs. As a result, cluster configurations output by scc for ProductSearch and PhotoShare result in 2x–4.5x average savings in cost compared to similarly performant scale-out options.

3.2 Problem setting and overview

Identifying an appropriate cluster architecture to host a large-scale service is often not straightforward. For example, given a set of resources to choose from (e.g., as shown in Table 3.1), an application provider has to answer several questions. What
storage technologies should be employed, and how should data be partitioned across them? Where should caching be employed? What types of servers should be chosen to house the selected storage units? In addition, even if the application’s implementation is efficient and there is coarse-grained parallelism in the underlying workload, how will algorithmic shifts in the application or variations in workload affect the appropriate cluster architecture? Our goal is to automate the process of answering these questions, rather than relying solely on human judgment.

**Problem setting.** In developing scc, our focus is on the typical scenario where a cluster is dedicated to a specific application, rather than large-scale data centers (e.g., Google, Microsoft) that host a mix of applications. scc caters to the common case where an application provider either acquires hardware or uses third-party infrastructure to deploy an application. In such cases, the question we seek to answer is: what information from the infrastructure provider and from the application developer is necessary to determine a cost-effective cluster configuration that meets performance goals?

**Overview of scc.** As shown in Figure 3.1, scc takes three inputs: i) a model of application behavior, specified by the application’s developer, ii) characteristics of available hardware building blocks specified by the infrastructure provider, and iii) application performance metrics, i.e., a parameterized service level agreement (SLA). Given these inputs, scc computes how cluster cost varies as a function of SLA and outputs a low-cost cluster configuration that meets the SLA at each point in the space. For example, a webservice SLA might specify a peak query rate per second. For each potential SLA value (e.g., 1000 queries per second), scc determines a cost-effective cluster architecture capable of satisfying the SLA. scc’s output cost vs. SLA value distribution helps administrators decide what performance can be supported cost effectively.
Our focus in developing scc is to show how to systematically exploit storage diversity; i.e, select among different physical media, local and remote storage, and various caching strategies. In the future, we plan to extend scc to tailor network configurations and choose among CPU types. Here, we assume the cluster network can deliver uniform bandwidth between all pairs of servers [7] and do not address incast-like scenarios [107] that arise due to limited packet buffers. Instead, we assume network storage access is limited only by network adapter speeds.

### 3.3 Inputs to scc

We now describe how we represent the three inputs to scc—SLA specifications, properties of cluster building blocks, and application models. Rather than model the intricate complexities of algorithms and hardware, scc captures aggregate high level statistics that are relevant to application and hardware scaling behavior over a broad range of scenarios. Towards this end, we identify a key set of elements that comprise each of scc’s inputs and the corresponding attributes required to describe these elements. Figure 3.2 depicts examples of scc’s three inputs; our implementation encodes them in XML.

#### 3.3.1 Specifying SLAs

We consider throughput-based SLAs for two distinct application classes: batch and interactive; we defer supporting latency-based SLAs to future work. For batch applications, the SLA has two attributes—the job size and the required execution time, e.g., for a MapReduce job, the SLA specifies the number of records to be processed and the total run time for doing so. scc is more applicable for provisioning a new set of VMs for every job than for provisioning a shared cluster used for running jobs with varying I/O and compute characteristics. For interactive applications run as services, each type of request is associated with its own performance-based SLA that describes its required sustained processing rate. For example, in the case of a photo sharing Web service, the rates of photo uploads, photo views, and album views are each specified as a separate SLA. scc’s SLAs specify peak rather than average case throughput. We discuss how scc
Figure 3.2: Example specifications of (a) SLAs for PhotoShare, (b) hardware resources, and (c) application behavior for a particular deployment of ProductSearch.

accounts for temporal variation in Section 3.6.3.
3.3.2 Cluster building blocks

`scc`’s second input is a characterization of the set of building blocks available for assembling the cluster. We account for three types of elements—storage units, CPU cores, and servers. To ensure our approach is not tied to the characteristics of any particular technology, we employ abstract features such as I/O bandwidth and number of processor slots as the attributes for these elements. Table 3.1 lists sample building blocks used in our evaluation.

**Storage**

Storage resources come in discrete units, e.g., 1 disk or 1 stick of DRAM. To differentiate between different kinds of storage technologies such as disk, SSDs and DRAM, we characterize each unit based on two properties: capacity and I/O throughput. Capacity is simply the amount of available storage measured in bytes. Representing I/O throughput is more complex; we capture it with four attributes—the average rate at which I/O requests are served and the average latency gap between serving successive I/O requests, accounting for both separately for reads and writes. The gap parameter captures overheads involved with non-sequential I/O, e.g., seeks on disks and block erasure on SSDs. We define read (write) gap for a particular storage device as the latency incurred on average between successive reads (writes) to random logical addresses on the device. The latency to serve a read (write) request for a chunk of size bytes is thus \((\text{size} \div \text{rate} + \text{gap})\). We consider gap rather than the commonly used IOPS metric because gap enables us to better capture the range of I/O performance regions from small to large records. For example, characterizing read performance on a 7.2K-RPM disk based on IOPS and rate works well for 4 KB and 10 MB reads, but fails to capture the read throughput with 200 KB reads. In our evaluation, we find that these four attributes—rate and gap for reads and writes—suffice to capture the I/O performance of multiple disk types and SSDs. Furthermore, we believe these attributes are expressive enough to capture the characteristics of phase change memory (PCM) and other emerging storage technologies.

The application-visible performance of a storage medium is also influenced by how the chosen file system places data. For example, a disk can deliver significantly
higher write throughput when written to in a log format [115]. Therefore, when an application stores a dataset on a storage or file system, we measure I/O rates and gaps of each storage unit when using that system to read/write data. Further, for each storage unit, we consider two other attributes: storage persistence (i.e., whether it provides non-volatile storage) and I/O bus type (e.g., SAS vs. PCIe).

**Servers and compute**

Servers impose constraints on how storage can be packed into a physical box. For each kind of server, we consider its memory capacity as well as the properties of its I/O controllers. For each I/O controller, we consider the total number of units it can support and its maximum available I/O bandwidth. For example, a serial attached SCSI (SAS) controller permits up to 128 connected disks, yet supports a maximum I/O bandwidth of only 6 Gbps, less than the total sequential I/O throughput that can be obtained from 128 disks. Similarly, throughput for remote storage is limited by a server’s network interface speed.

As our focus is on storage complexity in cluster architectures, we consider only a single CPU type, ignoring trade-offs in compute per unit power [10, 24]. Instead, we vary the number of cores per server to extract the level of parallelism needed to maximize storage utilization.

**Costs**

Finally, an additional attribute for every element in the resource specification is the amortized cost per hardware unit including both capital and operational outlays. In our current implementation, the latter subsumes energy costs, ignoring data center costs and administrator salaries, and we consider total cluster cost to be a linear sum of individual components, which may not necessarily be true for large quantities. We leave for future work discounting the growth of expenses with cluster size and accounting for increased operational costs with a higher diversity of server configurations in the cluster.
Figure 3.3: Interaction between tasks and datasets in example application PhotoShare. Edges between tasks and datasets represent I/O with direction differentiating input and output. Dotted edges indicate task dependencies.

### 3.3.3 Characterizing applications

Our characterization of applications accounts for two aspects—its implementation and the workload in its planned deployment. However, unlike previous attempts at formally specifying workloads [137], simply accounting for storage capacity needs and the application’s stream of I/O operations does not suffice for our purpose. Instead, to capture an application’s implementation, we first ask the application’s developer to describe its decomposition into compute and storage components, and the interaction between them. For example, Figure 3.3 depicts the components, and the interaction between them, for one of the three applications we consider later in our evaluation—a photo sharing Web service, PhotoShare. Though our approach places the onus on application developers to go through the process of formally specifying the components of their application, an application’s specification is reusable across deployments. Some of the characteristics of several applications are already captured today [97, 98].

Second, we enable those who deploy an application to annotate the specification of the application’s architecture with properties of the expected workload in their deployment. To do so, we require that the compute and I/O characteristics of an application’s components, when subjected to the target workload, be determined by running small-scale application benchmarks. Extracting these properties requires tracing the application’s execution—now standard practice in resource-intensive performance-critical
applications. In the absence of built-in tracing support, systems like Magpie [13] can be leveraged.

Tasks and datasets

scc’s application specification separates the application’s compute and storage requirements into tasks and datasets. A task is a specific application functional unit; all threads/processes that perform the same function together constitute a single task. A dataset is a collection of records of the same type with similar I/O access patterns.

Execution of tasks. To account for how compute time and I/O wait time are distributed across a task’s execution, we represent each task by its execution path; different tasks in an application will have different execution paths. A task in an interactive application executes its execution path for each incoming request, whereas in batch applications, a task’s execution path is executed as many times as necessary to consume its input. Further, since batch jobs can go through multiple phases of execution, we require the application developer to tag each task with the phase to which it belongs. The cluster can thus be provisioned to support the maximal resource requirement across phases.

We characterize the execution path of a task as a sequence of three types of operations—compute, I/O, and invocations of other tasks. Each of these can be marked as either blocking or non-blocking. Compute operations are characterized by the amount of time spent performing computation on a particular type of CPU. While this value can of course vary, we have found that a representative average is sufficient to inform scc; we show later in Section 3.6.1 that scc can help evaluate the sensitivity of its output to the input values. I/O operations are attributed with the dataset on which the operation is being performed and whether it is a read or write operation. Similarly, every task dependency is annotated with the invoked task.

The operations in a task’s execution path may not be completely deterministic. For example, an I/O operation may hit the cache in some cases but not all, or a remote task may need to be invoked only based on the results of prior task invocations. To capture such non-determinism, every operation has an additional attribute—the probability of its execution. This, for example, enables us to capture developer knowledge of typical working set sizes for individual datasets and the hit rate on the working set.
Lastly, we also require that each task node be tagged with its memory requirements. While some applications may use all available memory and garbage collect on demand, we consider required memory to be the amount necessary to maintain performance. Note that this specifies memory that \textit{scc} must allocate for computation beyond any additional DRAM \textit{scc} provisions as RAM disks to store datasets.

**Representing datasets.** Next, we account for datasets in terms of their I/O bandwidth and capacity requirements. The I/O requirements from a dataset are determined by all the I/O operations performed on it, across the execution paths of all tasks. We ask that each I/O operation be tagged with three attributes—the number of records read or written, the number of bytes in each record, and whether records are read in parallel. The last of these three properties can be specified by the application developer, while the other two depend on the workload for which the application is being deployed. Again, we find that average values suffice for our target throughput-based SLAs. Describing I/O in terms of records accounts for the overhead seen between successive read/write operations on storage media such as disks and SSDs, e.g., from disk seeks. We similarly annotate task dependencies with three attributes—the number of invocations being performed, whether they are in parallel, and whether the whole dependency is blocking or non-blocking.

Lastly, we account for a dataset’s capacity requirements by requiring that it be tagged with three additional attributes: its size, whether it must be persistent, and whether the dataset is local or remote. This last attribute differentiates between data assumed in the application’s implementation to be on a storage unit local to the task accessing it as opposed to data that may be stored on a storage unit on a different machine in the cluster. Though a remote file can be made to appear local by use of systems such as NFS, we capture the application developer’s assumption of local storage, since remote access leads to higher access latencies. \textit{scc} leverages this distinction in two ways. For a remote dataset, \textit{scc} explicitly accounts for network load resulting from I/O requests and some CPU requirements for the machines hosting the dataset. Conversely, task-local storage constrains the amount of parallelism available on a single machine due to the storage bandwidth and number of storage unit slots available on the node.

Figure 3.2(c) presents an example (for another of the applications we use in our
evaluation, ProductSearch, a product search Web service) of the precise format in which such an application characterization is specified as input to \textit{scc}.

### 3.4 Implementation of \textit{scc}

Next, we describe how \textit{scc} processes its inputs to generate cost-effective cluster configurations.

#### 3.4.1 Overview

\textit{scc} determines the cost versus SLA distribution for a given application deployment by considering the configuration for each point in the distribution independently. To compute the cluster configuration for a target SLA, \textit{scc} needs to answer two questions. First, it needs to determine the architecture of the cluster—for each dataset of the application, it must determine the type of media on which the dataset should be stored and how to pack the storage units into servers. This packing is constrained by the number and location of CPUs available to assign to the compute tasks that access each dataset. Second, \textit{scc} needs to identify the scale at which this architecture must be instantiated to meet the SLA—scale is determined by the number of servers, storage units, and CPUs, as well as the level of parallelism of each application task.

**Guiding Principles.** Two key principles help \textit{scc} identify the right cluster configuration. First, the architecture and scale for every application component can be determined independently when all operations are performed asynchronously, but not when some operations are synchronous. The SLA for any task only specifies the rate at which a task’s execution path must run. In the typical case where a task’s execution path contains some operations that block others, \textit{scc} needs to determine the “division of labor” across these operations that minimizes cost. For example, in a task that reads from an input dataset and then writes to an output dataset, in order to meet the task’s SLA, it may suffice to provision fast storage for any one of the two datasets; provisioning fast storage for both datasets may unnecessarily result in higher cost due to storage capacity requirements, whereas slow storage for both may incur higher costs in satisfying I/O throughput needs. Hence, \textit{scc} jointly determines resource requirements across
all application components.

Second, since \texttt{scc} is provisioning for peak load, it prevents over-provisioning by ensuring that at least one resource is bottlenecked on every server at peak load. (If the application provider desires to run the cluster at lower peak utilization, that can be specified as input.) Based on our characterization of hardware, there are four possible bottlenecks on each server—1) the number of slots or 2) the bandwidth on an I/O controller, 3) the number of CPU cores, or 4) network bandwidth.

\begin{figure}[ht]
\begin{center}
\begin{tabular}{|l|}
\hline
\textbf{Configuration state:} \( S = (S_1, \ldots, S_n) \), where \( S_i \) = storage type assigned to \( i^{th} \) dataset \\
\multicolumn{1}{|l|}{\textbf{for every remote dataset} \( d_i \)} \\
\quad compute \( U_i = \text{no. of units of } S_i \) to meet capacity and I/O needs from \( d_i \) \\
\multicolumn{1}{|l|}{\textbf{for every task} \( t_i \)} \\
\quad \( R_i = \text{average runtime of } t_i \) \\
\quad \( P_i \) (parallelism of task \( t_i \)) = \( \text{SLA}(t_i) \times R_i \) \\
\multicolumn{1}{|l|}{\textbf{for every dataset} \( d_j \) local to \( t_i \)} \\
\quad \text{compute no. of units of } S_j \text{ to meet capacity and I/O needs from } d_j \text{ for one instance of } t_i \\
\hline
\textbf{Linear integer program to choose servers} \\
\textbf{Variables:} \\
1. booleans for whether \( k^{th} \) server is of \( j^{th} \) type \\
2. \( \forall \) remote dataset \( d_i \), no. of units of \( S_i \) in \( k^{th} \) server \\
3. \( \forall \) task \( t_i \), no. of instances on \( k^{th} \) server \\
\textbf{Constraints:} \\
\textbf{Per-server constraints:} \\
1. On each I/O controller, (no. of storage units < no. of slots) \\
\quad \text{and (I/O throughput < bus bandwidth)} \\
2. (I/O throughput on remote datasets and local datasets \\
\quad \text{accessed remotely) < network bandwidth} \\
3. no. of CPUs < no. of CPU slots \\
\textbf{Per-dataset and per-task constraints:} \\
1. \( \forall \) dataset \( d_i \), (no. of units across all servers = \( U_i \)) \\
2. \( \forall \) task \( t_i \), (no. of instances across all servers = \( P_i \)) \\
\textbf{Objective: Minimize cost of (servers + storage units + CPUs)} \\
\hline
\end{tabular}
\end{center}
\caption{Summary of \texttt{scc}’s procedure for determining a cost-effective cluster configuration that satisfies target SLAs, given a particular assignment of storage types to datasets.}
\end{figure}

\textbf{Algorithm.} Driven by the need for joint optimization across components, \texttt{scc} represents each point in the state space of configurations by the assignment of storage unit types to datasets. As a result, if \( S \) is the number of storage choices and \( D \) is the
number of datasets, \textit{scc} has to search through a space of \(O(S^D)\) configurations; for each dataset, \textit{scc} can choose any one of the \(S\) storage options.

In cases where the configuration space is too large to perform an exhaustive search, \textit{scc} performs a repeated gradient descent search: We start with a randomly chosen configuration. In each step, we consider all neighboring configurations—those which differ in exactly one dataset’s storage-type assignment—and move to the configuration that still meets the SLA with the maximum decrease in cost. We repeat this step until we find a configuration where all neighbors have higher cost. Since gradient descent can lead to a local minimum, we repeat this procedure multiple times with different randomly chosen initial configurations and settle on the minimum cost output across the multiple attempts. In our evaluation, we have found that repeating the gradient descent 10 times is typically sufficient to find a solution close to the global minimum. Therefore, even when determining the configuration to satisfy workloads of tens of thousands of queries per second, \textit{scc}’s running time for any particular SLA is within a minute.

At the heart of \textit{scc}’s search of the configuration space is a procedure—described in Figure 3.4—that, given any particular assignment of storage types to datasets, determines a cost-effective set of resources to meet the target SLAs. In this procedure, \textit{scc} first determines for each remote dataset, i.e., not local to any task, the number of storage units required of the type assigned to the dataset in the configuration state. Second, \textit{scc} determines the number of CPUs required by every task and the number of storage units of the assigned type needed by the task’s local datasets. Finally, it determines the types of servers and number of each kind required to minimize overall cluster cost. We describe these three steps using examples from illustrative applications.

### 3.4.2 Resources for datasets

A dataset’s storage resources need to satisfy two requirements: capacity and I/O throughput. To determine the cheapest storage solution that satisfies both, \textit{scc} computes the number of storage units required to satisfy each requirement independently and chooses the maximum of the two. When the former (latter) is more expensive, we call the dataset capacity (I/O) bound. A capacity-bound dataset requires storage equal to the dataset’s size irrespective of the medium used. Determining the storage required
by a I/O-bound dataset is more involved. Though the total capacity of the storage units allocated to the dataset need only be equal to the dataset’s size, we may need more units—under-utilizing the capacity on each of them—to meet throughput demands.

We compute I/O requirements as follows. As described in Section 3.3.3, the application characterization specifies the record size and the number of records read/written in every I/O operation. \( scc \) computes the overall number of I/O operations that a particular storage unit can support based on its rate and gap parameters. The SLA combined with the probability attributed to an I/O operation fully specifies the required frequency of the operation, which in turn determines the number of storage units required to deliver the performance in parallel.

For example, when serving requests to view photos in PhotoShare, one photo of size 200 KB on average is read from the photos dataset on every photo view. If the photos dataset were assigned to 15K-RPM disk (Table 3.1), which offers a read rate of 150 MBps and a read gap of 3.5 ms, it will be able to serve 200 KB-sized reads at the throughput of \( \frac{200\text{KB}}{150\text{MBps}} + 3.5\text{ms} \), approximately 40 MBps. Therefore, if the SLA specifies 1000 photo views per second, \( \frac{200\text{KB} \times 1000/\text{s}}{40\text{MBps}} = 5 \) units of 15K-RPM disks are required to satisfy the I/O throughput requirement.

**Task phases**

Not all tasks in an application execute concurrently, e.g., the Map and Reduce tasks run in different phases of a MapReduce job. Since datasets are subject to I/O operations only from tasks executing in a particular phase, \( scc \) computes the storage needed to meet I/O requirements in each phase independently. The storage requirements for a dataset during a particular execution phase are computed as the sum of storage needs across all the I/O operations made on the dataset by the tasks that run in that phase. \( scc \) computes the overall I/O-mandated storage requirement as the maximum over all phases.

**Caching for higher I/O**

When a dataset is I/O-bound, storing it across units of a single type may not always be the cheapest solution. I/O throughput of persistent datasets can be improved
by introducing a second type of storage unit as a caching layer. For example, when considering a single storage type to service the entire load, the SSD is the most cost-effective option for the tags dataset in the PhotoShare application. However, a cheaper solution is to store the persistent copy of the tags on a 7.2K-RPM disk and to serve reads from a cached copy in DRAM.

scc assumes write-through caching. Persistent storage units handle all writes and maintain a persistent copy. Units of another type, with higher I/O rates, handle all reads. To ensure durability, every write is committed to both copies and by default, scc provisions enough storage to cache the entire dataset. However, developer knowledge of the application’s working set size—encoded into the application specification as different capacity requirements for the dataset and for the cache—can also be used to determine what fraction of the dataset is to be cached. To evaluate whether such a solution is cost effective, scc computes the costs of both copies of the dataset separately and computes their sum.

3.4.3 Task Resources

scc next determines the resource requirements of each compute task in three steps. First, it determines the CPU utilization of the task. Second, it computes the degree of parallelism—i.e., the number of threads/processes of the task—required to meet the SLA. Finally, it determines the number of storage units required per instance of the task for each of the task’s local datasets.

A task’s CPU utilization is the fraction of its run time spent performing computation. scc translates a task’s CPU utilization into the corresponding CPU resources required by computing the level of parallelism required to meet the SLA: if a task’s execution path is to be executed with frequency $F$ and the task’s average run time is $R$, then $(F \cdot R)$ instances of the task are required. The value of $F$ for a task is computed from the SLA for that task and other tasks that depend on it; $R$ is computed by appropriately summing up the times for compute, I/O, and task invocation operations in the task’s execution path, taking into account, for each operation, its probability and whether it is blocking or non-blocking.

scc calculates each task’s storage requirements for its local datasets based on
capacity and I/O throughput requirements. \textit{scc} also computes the task’s memory requirements and the network bandwidth needed for I/O accesses to remote storage. \textit{scc} determines these three requirements—local storage, memory, and network throughput—per instance of the task and linearly extrapolates to a target level of parallelism.

### 3.4.4 Optimizing server costs

Finally, \textit{scc} optimizes cluster cost by minimizing the cost of required servers. Determining the servers required to host storage and CPU resources reduces to the multi-dimensional vector bin packing problem [29]. Each server type is associated with a cost and a vector of resource limits, such as the I/O bandwidth of each I/O controller and the maximum number of CPUs that the server can accommodate. Respecting these resource limits, CPUs and storage units required by tasks and datasets must be placed across servers, while minimizing total cost. \textit{scc} solves this bin-packing problem with a linear integer program.

### 3.5 Evaluation

Next, we demonstrate that \textit{scc} achieves the right cost versus performance trade-off. Unfortunately, it is difficult to select appropriate comparisons. Though there exists a large body of work on capacity planning [85], all of it revolves around the question: “Given a cluster architecture for an application, how many servers of each type in the architecture are necessary?” In contrast, \textit{scc} minimizes cost by determining not only the right scale, but also the architecture most suited for a given application deployment. Moreover, conversations with major infrastructure providers reveal that existing approaches for provisioning cluster applications used in practice are ad-hoc—the primary motivation for our work.

#### 3.5.1 Methodology

We apply \textit{scc} to three distributed applications with disparate workload characteristics to identify the cost-versus-SLA tradeoff in each case. To keep the discussion
simple, we fix capacity requirements while varying the SLA. For each application, we validate the cost-effectiveness of \textit{scc}'s output for one particular target SLA. Though \textit{scc} readily outputs cluster configurations on the scale of tens of thousands of servers, we focus on smaller scales for validation so that we can instantiate the configurations with hardware we have on hand. Note that even at the scale of a few servers, the combination of type and quantity for storage, compute, and servers results in a very large configuration space. For example, with 5 servers of type Server1, over $10^{14}$ cluster configurations are feasible using the building blocks in Table 3.1.

In the absence of prior approaches for principled determination of cluster architectures, our evaluation compares configurations output by \textit{scc} with \textit{all} possible alternative assignments of datasets to storage types; for each alternative, we consider those quantities of hardware resources to make cost comparable to \textit{scc}. Here, we present results from alternate architectures that come closest to matching \textit{scc} with respect to satisfaction of SLAs. In some cases, we also consider alternative architectures at the scale required to meet input SLAs and show that they incur higher costs than \textit{scc}. For each experiment, we physically provision clusters composed of the building blocks provided as input to \textit{scc}.

Table 3.1 summarizes the resources provided as input to \textit{scc}, represented formally as in Figure 3.2(b). We construct our specification for cluster building blocks based on HP ProLiant DL380 G6 servers interconnected by a Gigabit Ethernet network. In each server (Server1), we consider the resource limitations to be one quad-core Intel Xeon processor, four SAS slots, and up to 12 GB of DRAM. Each of the SAS slots can support a 7.2K-RPM disk, a 15K-RPM disk, or an Intel SSD. To evaluate the performance of a given configuration, we turn off CPU cores and/or use only a subset of the SAS and DIMM slots.

For each of the resources, we consider the cost to be the amount we paid, excluding support, plus energy costs computed based on power usage numbers from product data sheets (we assume $0.10/kWh$ over a three year deployment). Though the power drawn by any unit can vary from its specification, we study the robustness of our results (Section 3.6.1) and find that they remain unchanged even if energy costs increase by a factor of two.
Figure 3.5: Validation of cluster output by \textit{scc} for particular SLA values in the three application cases.

3.5.2 Photo sharing

Our first application, PhotoShare, is an interactive photo sharing application. It allows users to upload tagged photos, to view thumbnails for photos associated with a given tag, and to view the photos. PhotoShare is a C++ FastCGI application hosted on
Figure 3.6: Cost versus SLA distribution output by *scc* for PhotoShare. Note log scale on y axis.

lighttpd webservers. Uploaded images are thumbnailed and stored, whereas tag updates are made via RPCs. Data is kept in a distributed log-based key-value storage system. Image, tag, and thumbnail views translate to fetches from the store. The three SLA metrics are the simultaneous rates for uploading photos, viewing photos, and viewing thumbnails associated with tags. Our input workload has, on average, 200-KB images that convert to 4-KB thumbnails, and an average of 10 photos/tag and 10 tags/photo.

We apply *scc* to study the cost as a function SLA by fixing the ratio of the rates for uploads, photo views, and tag views at 1:3:1. Figure 3.6 shows this cost distribution for a range of SLA values. Perhaps surprisingly, no huge spikes are observed in this distribution; this is because *scc* balances costs across the kind of storage, the number of CPUs, and the number of machines provisioned. Rather than adding more machines of the same type, the cluster architecture transitions to faster storage as the SLA becomes more stringent, with transitions in storage type for different datasets seen at different SLA values. Table 3.2 highlights these transitions. Note that the quantity in which different types of resources are provisioned varies within each architecture regime specified by every row in the table.

We further compare the cost output by *scc* with the cost associated with a scale-out approach. We compare the *scc* configuration to the cases where the building block is based around: 1) storage servers with four 7.2K-RPM disks (the cost-optimal storage type for all datasets at the lowest SLA), and 2) servers with four 15K-RPM disks. In either case, more storage servers are added as the required rates increase. Figure 3.6
Table 3.2: Different regimes based on SLA requirements in the cost-effective architecture for PhotoShare.

<table>
<thead>
<tr>
<th>Uploads/s</th>
<th>Photos</th>
<th>Thumbnails</th>
<th>Tags</th>
</tr>
</thead>
<tbody>
<tr>
<td>≤ 5</td>
<td>Disk</td>
<td>Disk</td>
<td>Disk</td>
</tr>
<tr>
<td>5–25</td>
<td>Disk</td>
<td>Disk</td>
<td>Disk + DRAM</td>
</tr>
<tr>
<td>25–330</td>
<td>Disk</td>
<td>SSD</td>
<td>Disk + DRAM</td>
</tr>
<tr>
<td>330–930</td>
<td>SSD</td>
<td>Disk + DRAM</td>
<td>Disk + DRAM</td>
</tr>
<tr>
<td>930–10k</td>
<td>Disk + DRAM</td>
<td>Disk + DRAM</td>
<td>Disk + DRAM</td>
</tr>
</tbody>
</table>

shows that the costs in both cases are significantly greater than with scc, incurring between 3 and 4.5 times more cost (note the logarithmic y axis). Thus, simply scaling out a homogeneous configuration that is cost-effective at low loads can result in significant cost inflation at higher loads.

To verify the performance of scc’s suggested configuration, we focus on one particular SLA: 100 uploads/s, 300 photo views/s, and 100 tag views/s. The fraction of the SLA satisfied is the minimum fraction of sustained request rates across uploads, photo views, and tag views. scc determines the following cluster configuration for this SLA: one machine, with 4 CPU cores and 2 GB of DRAM hosts the webserver; a second machine stores the photos across four 15K-RPM disks; and a third machine hosts one SSD for thumbnails, and 1 GB of DRAM and one 7.2K-RPM disk for tags. Each of the two storage machines have 2 CPU cores and an additional 1 GB of DRAM, as required by the key-value storage system.

Figure 3.5(a) shows that this configuration meets the SLA; in fact, the configuration is slightly over-provisioned. It also shows the configuration is near a minimum: removing a core from the webserver (Alt1), replacing the thumbnail’s SSD with a cheaper 15K-RPM disk (Alt2), removing one of the photo disks (Alt3), or replacing the thumbnail’s SSD with two 7.2K-RPM disks (Alt4) all result in SLA misses. A scale-out architecture extending Alt4 with more 7.2K-RPM drives (Alt5) incurs 30%-higher cost to meet the SLA.
3.5.3 Product search

Our second application is a multi-merchant product search and comparison service, which we call ProductSearch. We store product tables, which include product serial numbers, types, descriptions, and costs, along with product type field indices in a Hadoop Distributed File System (HDFS). In addition, user rating data is stored in a separate database table. Worker processes running across the cluster process queries for the cheapest product of a given type with a minimum user-specified rating. Each worker maintains a local copy of the ratings table as well as an index on the product serial number field; the ratings table and index are hence, specified as local datasets in the application’s specification. To execute a query, a worker fetches the relevant product table and index from HDFS and then performs a join with the ratings table on the product serial number field, selecting for rows with the specified product type.

In our deployment, we build product tables with an average of 200K products, each with an average of 200 ratings. This translates to 8 GB for the ratings and roughly 800 MB for each product table. The SLA for this application specifies the required query rate.

We apply scc to determine system cost as a function of the SLA value. As with PhotoShare, the architecture of the cost-effective cluster changes significantly across different regimes of the SLA. At low query rates, scc recommends disks for both HDFS and local storage of workers. As the required query rate increases, scc transitions to using faster storage or provisioning more machines to handle the increased load. Figure 3.7 illustrates one particular transition between query rate regimes. Also, in this case as well, scc’s configurations yield significant cost savings compared to simple scale-out options—roughly $3 \times$ and $2 \times$ savings on average in comparison to the scaling out of homogeneous configurations with 7.2K-RPM and 15K-RPM disks, which are cost-optimal at low loads.

We validate scc with an SLA of 12 queries per minute. scc’s cluster output for this case has two parts. First, the HDFS repository is stored across two machines, each with one CPU and two 7.2K-RPM disks. Second, 12 worker processes are spread across three machines, each with one CPU and four SSDs. We run this configuration for 15 minutes. Figure 3.5(b), which plots the fraction of required queries completed during
the experiment, shows that this configuration is able to meet the SLA.

Next, we compare scc’s output with alternative configurations. First, we consider clusters with alternative local storage for the workers—Alt1 and Alt2 use 15K-RPM drives, and Alt3 uses 7.2K-RPM disks with DRAM. In each case, we consider the number of workers and servers to keep cost comparable to scc. In both Alt1 and Alt2, the disk’s lower random read throughput inflates query processing times and, hence, aggregate throughput falls well below the SLA. The performance of Alt3 comes close to the SLA, but still falls short. Second, when we place all four disks underlying HDFS into one machine (Alt4), the 1 Gbps network becomes a bottleneck relative to the aggregate read throughput from four 7.2K-RPM drives. As a result, download times increase, leading to SLA violations.

We also use this example application to test scc’s ability to capture knowledge of working set sizes. We again apply scc to satisfy the SLA of 12 queries per minute, but this time with the additional input that 20% of product types receive 80% of queries (the application specification for this case is shown in Figure 3.2(c)). In this case, scc outputs an alternate architecture where 12 worker processes, previously run on three machines each with four SSDs, are now instead run on three machines each with four 15K-RPM disks and 10 GB of DRAM. Queries to “hot” products are served from DRAM and those to “cold” data are served from the disks. This configuration meets the SLA with 7%-lower cost than the case where access patterns were assumed to be uniform.
3.5.4 Sorting binary tuples

Our final application, Terasort [100], is a MapReduce job that sorts collections of 100-byte tuples, each consisting of a 10-byte key and a 90-byte value. A Mapper reads tuples from a local input file and sends them over the network to appropriate Shuffle processes. Each Shuffler writes the tuples it receives to a set of intermediate, sorted local files. Once the Mappers and Shufflers are done, the Shuffle processes transform into the role of Reducers. Each Reducer merges the tuples in the local files into an output file of sorted tuples. For this application, the SLA is the total runtime of the MapReduce job.

We use scc to determine the cost of clusters capable of sorting 50 GB for a range of runtimes. Note that though we put together clusters of individual servers here, we envision that scc will be used for such jobs to provision a set of virtual machines in a virtualized infrastructure. Unlike PhotoShare and ProductSearch, we see no significant architecture changes over different runtimes. scc uses the basic building block of provisioning Mappers on machines with four cores and one 7.2K-RPM disk and Shufflers/Reducers on machines with four cores and two 7.2K-RPM disks. scc provisions more machines for both components to meet more stringent SLAs. Faster storage has no benefits because the job is CPU bound.

Next, we verify the performance of the cluster output by scc for an SLA that requires 50 GB to be sorted in 25 minutes—an average sorting rate of 2 GB per minute. The scc cluster consists of 8 Mappers and 16 Reducers spread across two and four machines respectively with the above-mentioned building blocks. We run the application on this cluster to sort 50 GB of input data. Figure 3.5(c) plots the SLA-specified runtime divided by the observed runtime and shows that the scc cluster meets the SLA.

To evaluate the cost-effectiveness of scc’s output, we also sort 50 GB of data on several alternative architectures. A few such alternatives include Alt1 and Alt2, which reduce the number of cores from 4 to 3 on the Mapper machines and on the Reducer machines, respectively. Alt3 substitutes the two 7.2K-RPM disks on each of the four Reducer machines with one 15K-RPM disk shared between the intermediate and output data. Figure 3.5(c) shows that the runtime of the Terasort job misses the SLA by at least 10% in every case. The figure also shows that two other alternatives—Alt4 and
Alt5—which have similar cost to scc’s output but trade off compute resources for more or faster storage, also fall short.

Unlike our other two example applications, compute-intensive MapReduce jobs have a cluster configuration recommended by best practices. We modify the cluster architecture to be six machines each with four cores and two 7.2K-RPM disks—a setup recommended by Cloudera for a “Balanced Compute Configuration” [32]. Also, we configure every node in the cluster to run a fixed number of Mappers and Reducers. We evaluate three different combinations of Mappers and Reducers per node (the “2M 2R”, “2M 3R”, and “1M 3R” points in Figure 3.5(c)), and interestingly, we find that the recommended MapReduce configurations deliver lower performance than scc for similarly priced clusters. While all three alternatives meet the SLA when scaled out to an additional machine, e.g., the “2M 2R+” point in the figure, this results in 16%-higher cost than scc’s recommendation.

3.6 Discussion

In this section, we discuss the robustness of scc’s output, its utility in planning application implementation architectures, and its extensibility on other fronts.

3.6.1 Robustness of scc’s output

scc’s output cluster configuration for a target SLA is a function of both the SLA and the exact values specified for the various attributes in the application and hardware specifications. In practice, a user of scc may not have precise values for all attributes due to incomplete knowledge of the application workload, uncertainty of hardware costs, or measurement inaccuracy in benchmarking.

scc is naturally built to cope with such uncertainty. For every attribute in the input specifications, scc varies the value of the attribute in the neighborhood of the initially specified value. For each attribute, it then outputs the range of values for that attribute wherein the cost-effective cluster architecture, i.e., the types of resources assigned to different application components, remains unchanged; variance of the attribute’s value within this range can be handled by simply adding more resources of the same type.
Table 3.3: Determining robustness of scc’s output with respect to its input: (a) robustness of cluster configuration with respect to input values for a sample set of attributes, and (b) the change in hardware costs to which scc’s storage decision for each dataset is most sensitive.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Range with same architecture</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Lowest value</td>
<td>Input value</td>
<td>Highest value</td>
</tr>
<tr>
<td>Avg. photo size</td>
<td>50 KB</td>
<td>200 KB</td>
<td>850 KB</td>
</tr>
<tr>
<td>Avg. thumbnail size</td>
<td>1 KB</td>
<td>4 KB</td>
<td>30 KB</td>
</tr>
<tr>
<td>SSD unit price</td>
<td>$200</td>
<td>$450</td>
<td>$900</td>
</tr>
</tbody>
</table>

(a)

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Most sensitive to what change in hardware costs?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Photos</td>
<td>20% drop in $ of 7.2K-RPM disk</td>
</tr>
<tr>
<td>Thumbnails</td>
<td>92% drop in $ of DRAM</td>
</tr>
<tr>
<td>Tags</td>
<td>31% drop in $ of 15K-RPM disk</td>
</tr>
</tbody>
</table>

(b)

Outside of that range, the cluster will need to be revamped with a different type of resource for some application component, a significantly more cumbersome undertaking. For example, we again consider PhotoShare with an SLA of 100 uploads/s, 300 photo views/s, and 100 tag views/s. Table 3.3(a) shows the value ranges output by scc for a few attributes, within which the cluster architecture is robust to change. For example, we see that as long as average photo size remains between 50 KB and 850 KB, the cluster architecture remains the same as that obtained with the input value of 200KB.

Furthermore, scc can also evaluate the sensitivity of its choice of storage configuration for every dataset in the application. For example, consider PhotoShare again with the same input SLA as above. Based on current hardware costs, scc determines that photos be stored on 15K-RPM disks, thumbnails be stored on SSDs, and tags be stored persistently on 7.2K-RPM disks and cached in DRAM, in order to meet the SLA at minimum cost. However, these recommendations are likely to change as prices for storage units drop. scc can determine how robust are its choice of storage options to such changes in hardware prices. To do so, it varies the price of every type of storage unit from its input value down to 0, and notes the inflection points at which the optimal stor-
age choice for some dataset changes. Based on this analysis, it can determine, for every dataset, that change in hardware price to which the current storage choice for the dataset is most sensitive. Table 3.3(b) shows the output of this analysis for the three datasets in PhotoShare. While the storage choices for photos and tags are sensitive to relatively small reductions in the prices for 7.2K-RPM and 15k-RPM disks, *scc*’s recommendation of storing thumbnails on SSDs is very robust to price fluctuations.

### 3.6.2 Informing application development

Thus far, we assumed a fixed application implementation. However, *scc* can also help determine the best application architecture. For instance, in the case of Terasort, there is a fundamental performance tradeoff between a cluster configuration with sufficient DRAM to store all data to be sorted and one that must stage portions of the data into memory from secondary storage. The former case requires one read and one write of all the data while the latter requires two reads and two writes of the data [5].

To explore cost–performance tradeoffs for the two application architectures, we must consider the benefits of servers with more network bandwidth (so remote storage does not become a bottleneck) and more memory (to allow for storing the entire dataset in memory). In Table 3.1, Server2 is the same HP ProLiant DL380 G6 server as Server1, but with more resources per server and a 10-Gigabit Ethernet (10GigE) NIC. Server3 is the HP ProLiant DL785 G5 Server, which accommodates more processors and DRAM, again with a 10GigE NIC.

We use *scc* to determine the cluster configuration necessary to sort 100 TB in the time required to read/write the whole data from/to disks twice at the read/write rate of the 7.2K-RPM disk. This cluster costs $239K and completes sort in 10,000 seconds. For the alternative implementation where all data fits in DRAM, we apply *scc* to satisfy the SLA of sorting the complete dataset in half the SLA of the baseline implementation. The cheapest cluster configuration determined in this case costs $5.6M and sorts 100 TB in 5,000 seconds. Thus, according to *scc*, the latter implementation provides a 2× speedup at 24× the cost. The application designer can decide if the faster processing is worth it.
3.6.3 Extensibility of scc

Our approach of determining cost-effective cluster configurations with scc is extensible in several ways.

**Less flexible infrastructure services.** Though we restrict our attention in this chapter to flexible infrastructure services that permit arbitrary mixing and matching of compute and storage resources on a per-server or per-VM basis, scc can also be readily applied to less flexible services that offer only certain combinations of processor, storage, and memory configurations, e.g., Amazon’s EC2 service [9]. In such cases, each combination of resources offered by the infrastructure service can be provided as input to scc as a separate server type, and the cost of each server will subsume the costs of all the resources that come with it.

**Accounting for availability.** Though we have focused on performance requirements of applications thus far, performance and availability SLAs need to be considered in unison. For example, a cheap disk type may be an attractive option for a capacity-bound dataset but the degree of replication necessary to meet availability goals may make the option cost-prohibitive. scc can be extended to pick for each dataset that combination of storage type and associated replication factor that meets the combination of performance, availability, and consistency requirements at minimum cost.

**Load variation and incremental growth.** Our current implementation of scc provisions applications for peak load. However, when the distribution of load across time is available, scc can leverage the information in two ways. First, scc can estimate energy costs more accurately. Second, when pricing for resources is “elastic”, i.e., a user can provision resources on-demand and pay for what she uses, scc can make incremental reconfiguration decisions, determining when to simply scale-out and when to switch between architectures. scc’s distinction between remote persistent datasets and local transient datasets enables it to capture the costs associated with data redistribution.

**Network configuration and CPU diversity.** scc’s specification of application behavior can be used to infer the communication pattern among the application’s components, and thus inform configuration of the cluster’s network. For example, in the case of ProductSearch, scc can infer from the application specification that the workers communicate only with the HDFS repository but not among themselves. scc can then
use this information to recommend a bi-partite network with servers hosting HDFS on one side and servers hosting workers on the other side. \textit{scc} can also be readily extended to choose among a range of CPUs; the application specification simply needs to include for every compute operation the time required for that operation on each type of CPU.

### 3.7 Related work

Our work builds upon and shares some similarities with several lines of prior work.

\textbf{Tuning storage:} Minerva [8], Hippodrome [11], and Rome [137] automate the provisioning of disk arrays with a similar approach of characterizing workloads and storage. Ursa Minor [1] varies erasure coding parameters depending on an application’s availability requirements. PADS [16] is configurable to build a wide range of replication systems with varying consistency semantics. In contrast to all of these efforts, we consider an application’s storage and compute requirements in unison. Moreover, we choose among different storage media such as disk, SSD, and DRAM to minimize cost, with multiple media possibly being used for the same application.

\textbf{Application modeling:} Bodik et al. [20] infer application performance models by applying machine learning techniques on statistics gathered by monitoring the application execution. Thereska et al. [125] predict performance across application configurations based on statistical models. IRONModel [124] corrects deviations between the performance of running systems and high fidelity models. In all cases, since application models are tuned to specific cluster configurations, they are not directly applicable to alternative hardware configurations.

Stewart and Shen [120] build performance models of multi-component applications to aid in the placement of application components on a given cluster. Osogami and Itoko [101] apply hill-climbing techniques to automatically determine web-server parameters, and Liu et al. [82] construct a queuing model for a three-tiered web service to predict throughput and response times. Again, all of these consider a fixed hardware configuration.

\textbf{Application-specific cluster architectures:} Application developers have con-
verged on a range of cluster architectures for individual applications. Several web services employ DRAM caches using distributed in-memory storage systems [44, 102]. Applications such as WER [53] use clusters that have separate sets of machines for compute and storage. FAWN [10] and Gordon [24] use SSDs to build performant yet power-efficient distributed data processing systems. MRPerf [132] and Starfish [63] use an approach similar to scc but focus solely on predicting cluster requirements of MapReduce setups. scc not only infers these cost-effective architectures for existing applications, but also enables the inference of the right cluster architecture for emerging applications.

**Storage and computing services.** There been a few recent attempts [65, 52] at satisfying SLAs in the setting of a compute and storage cluster shared across applications. Such multi-application environments have also seen the recent emergence of virtual storage appliances. scc is targeted at the still significantly more common scenario of cluster deployments for a single application.

### 3.8 Summary

In this chapter we explored how the deployment of applications on clusters is more cost-effective if informed by characterizations of application behavior and hardware properties. Towards this end, we presented how these inputs can be specified, and we developed scc to compile these inputs into cost-effective cluster configurations. Our experiments in applying scc to a range of application workloads and storage options show that scc captures sufficient detail to prescribe the right combination of storage and server hardware at the right scale; modifying the architecture or reducing the scale leads to significant performance degradation. To meet application demands, scc often predicts heterogeneous cluster architectures that result in significant cost savings compared to simply scaling out homogeneous architectures. scc provides the flexibility to explore variations in application behavior. This is especially useful because performance improvements in application behavior further enhance cluster efficiency. While scc is beneficial in the context of isolated applications and storage, it has not yet been extended to consider shared resources. In the next chapter, we present a new technique to improve
the performance for shared storage resources.

Chapter 3, in part, is a reprint of material as it appears in the article “scc: Cluster Storage Provisioning Informed By Application Characteristics and SLAs” by Harsha Madhyastha, John C. McCullough, George Porter, Rishi Kapoor, Stefan Savage, Alex C. Snoeren, and Amin Vahdat which appears in Proceedings of the USENIX Conference on File and Storage Technologies, 2012. The dissertation author was a primary investigator and co-author of this article.
Chapter 4

Stout

As we described in Chapter 1, the applications that are deployed on large clusters are frequently composed of smaller pieces. Common pieces, such as the storage layer, can be shared across multiple applications. The techniques we described in Chapter 3 consider storage for a single application, but do not allow for the benefits of multiplexing multiple applications. Multiplexing applications can lead to improved resource utilization and lower cost by aggregating the costs of provisioning for independent peak loads.

The drawback to the efficiency benefit of a shared resource is that the performance of a given application is now coupled to the other applications that use the same service. Scalable key-value storage systems are a popular way to store small snippets of data (e.g., your current Twitter feed), in a lightweight manner. Applications using key-value storage systems try to get the lowest latency possible by sending requests right away. As the number of incoming requests increases, the latency degrades for all applications using the same key-value storage system. Thus, an application sharing the same key-value system with a service that is currently sustaining high load will share the same degraded-latency fate. There are techniques that can improve the overall performance of the system in high load scenarios, but these techniques sacrifice low latency in low load scenarios.

In this chapter, we present Stout, a system that helps applications adapt to variations in storage layer performance by treating scalable key-value storage as a shared resource requiring congestion control. We use the insight that delaying storage requests
and batching writes together reduces the number of storage requests to the same key and allows for reduced protocol and transaction overheads. The challenge that we address is identifying an appropriate batching interval for the current storage system load. In this chapter, we present the challenges involved in improving performance with adaptive-batching, how we apply techniques from congestion control and networking to storage systems, and we evaluate our system’s performance on a real cloud service.

4.1 Introduction

Application developers are increasingly moving towards a software-as-a-service model, where applications are deployed in data centers and dynamically accessed by users through lightweight client interfaces, such as a Web browser. These “cloud-based” applications may run on hundreds or even thousands of servers to support hundreds of millions of users; the application servers in turn leverage high-performance scalable key-value storage systems, such as Google’s BigTable [27] and Microsoft’s Azure Storage [86], that allow them to gracefully handle variable client demand. Unfortunately, because these storage systems support many applications on a single shared infrastructure, they present application developers with a new source of variability: every application must now cope with a store that is being loaded by many applications’ changing workloads.

Unlike variability in its own workload, which an application can easily monitor and often even predict, changes in the level of competition for shared storage resources are likely to be unexpected and outside the control of a particular application. Instead, each individual application must observe and react to changes in available storage-system throughput. Ideally, the collection of applications leveraging a particular scalable storage system would cooperate to achieve a mutually beneficial operating point that neither overloads the storage system nor starves any individual application.

Today, each application seeks to minimize its own perceived latency by sending each storage request immediately. Each storage request thus incurs overheads such as networking delay, protocol-processing, lock acquisitions, transaction log commits, and/or disk scheduling and seek time. However, when the store becomes heavily loaded,
sending each request individually can lead to queuing at the store, and consequently high delay or even loss due to timeouts. In such heavily loaded situations, the throughput of the storage service can often be improved by batching multiple requests together, thereby reducing queuing delay and loss. Batching achieves this improvement by amortizing the previously mentioned overheads across larger requests, and prior work has documented that many stores provide higher throughput on larger requests [27, 59, 115].

Dynamically adjusting their degree of batching allows applications to achieve lower latency under light load and higher throughput under heavy load. Unfortunately, existing work applying control theory to computer systems offers no easily applicable solutions [66, 71]. For example, a common assumption in control theory is modest actuation delay: a reasonable and known fixed time between when an application changes its request rate and the store responds to this change. Scale-out key-value storage systems do not have such bounds, as an application can easily create a very deep pipeline of requests to the storage system. Other control theory techniques avoid this assumption, but bring other assumptions that are similarly unsatisfied by such storage systems. Instead, we observe that managing independent application demands in a scale-out key-value storage environment is quite similar to congestion control in a network: the challenge in both settings is determining an application’s (sender’s) “fair share.” Moreover, the constraints of distributed congestion control—multiple, independent agents, unbounded actuation delay, and lack of a known bandwidth target—are quite similar to our own. Hence, we take inspiration from CTCP [122], a recently proposed delay-based TCP variant which updates send-rates based on deviation from the measured round-trip latency.

We propose an adaptive interface to cloud key-value storage layers, called Stout, that implements distributed congestion control for client requests. Stout works without any explicit information from the storage layer: its adaptation strategy is implemented solely at the application server (the storage client) and is based exclusively on the measured latency from unmodified scalable storage systems. This allows Stout to be more easily deployed, as individual cloud applications can adopt Stout without changing the shared storage infrastructure. Stout both adapts to sudden changes in application workload and converges to fairness among multiple, competing application servers employ-
We show experimentally that Stout delivers good performance across a range of workloads requiring batching intervals to vary by over two orders of magnitude, and that Stout significantly outperforms any strategy using a fixed batching interval. Based on these results, Stout demonstrates that much of the benefit of adaptation can be had without needing to modify existing storage systems; to use a new store, Stout requires only internal re-calibration. By allowing cloud applications to sustain higher request rates under bursts, Stout can help reduce the expense of over-provisioning [28, 103]. Simultaneously, Stout provides good common case storage latency; this is critical to user-perceived latency because generating a user response often requires multiple interactions with the storage layer, thereby incurring this latency multiple times [39].

The primary novelty of Stout is its adaptive algorithm for dynamically adjusting the batching of storage requests. To better understand both the benefits and challenges in building an adaptive interface to shared cloud storage, we evaluate our adaptive control loop using a workload inspired by a real-world cloud service that is one component of Microsoft’s Live Mesh cloud-based synchronization service [87]. In our performance evaluation, we demonstrate that: 1) Stout successfully adapts to a wide range of offered loads, providing under light workloads over 34× lower latency than a long fixed batching interval optimized for throughput, and under heavy workloads over 3× the throughput of a short fixed batching interval optimized for latency; 2) Stout provides fairness without any explicit coordination across the different application servers utilizing a shared store; and 3) the same adaptation algorithm works well with three different cloud storage systems (a partitioned store that uses Microsoft SQL Server 2008; the PacificA research prototype [80]; and the SQL Data Services cloud store [88]).

### 4.2 Background

Stout targets interactive cloud services. This class of services requires low end-user latency to a variety of data. Stout facilitates high-performance storage access for these services by controlling and adapting the way the services make use of back-end key-value storage systems to provide the best possible response time (i.e., minimize
end-user latency). While we believe that Stout’s general approach of using a control loop to manage the interactions with a persistent storage tier holds promise for many different kinds of cloud-based services, including those that process large datasets (e.g., services that use MapReduce [38] or Dryad [67]) the rest of this section elaborates on our current target class of interactive latency-sensitive cloud services.

Stout works with scalable services that are partitioned. A partitioned service is one that divides up a namespace across a pool of servers, and assigns “keys” within that namespace to only one server at a given point in time. To enable fast response times, the objects associated with the partition keys are stored in memory by these servers. Stout is responsible for handling all interactions with the back-end persistent storage tier. Figure 4.1 depicts a typical three-tier cloud service, and where Stout fits within that model. The first tier simply consists of front-end Web servers that route end-user requests to the appropriate middle-tier server; the middle tier contains the application logic glued together with Stout, and the back-end tier is a persistent storage system.

As a concrete example, consider an online spreadsheet application, such as that
provided by Google Docs [55]. The user-interface component of the spreadsheet application runs inside the client Web browser. As users perform actions within the spreadsheet, requests are submitted to the cloud infrastructure that hosts the spreadsheet service. User requests arrive at front-end Web servers after traversing a network load balancer, and the front-end server routes the user request to the appropriate middle-tier server which holds a copy of the spreadsheet in memory. Each server in the middle tier holds a large number of spreadsheets, and no spreadsheet is split across servers. Whenever the processing of a user request results in a modification to the spreadsheet, the changes are persisted to a scalable back-end storage system before the response is sent back to the user.

Many of today’s Web services are built using the same paradigm as the spreadsheet application above. For example, a service for tracking Web advertising impressions can store many “ad counters” at each middle-tier server. Email, calendar, and other online office applications can also use this partitioning paradigm [55, 89, 90].

Forcing writes to stable storage before responding to the user ensures strong consistency across failures in the middle tier; that is, once the user has received a response to her request to commit changes, she can rest assured they will always be reflected by subsequent reads. So long as a middle-tier server maintains these semantics, it is free to optimize the interactions with the storage layer. Thus, when a middle-tier is handling multiple changes, it can batch them together for the storage layer.

### 4.3 Adaptive Batching

Batching storage requests together before sending them to the store leads to several optimization opportunities (Section 4.3.1). However, delaying requests to send in a batch is only needed when the store would otherwise be overloaded; if the store is lightly loaded, delaying requests yields a net penalty to client-visible latency. This motivates Stout’s adaptation algorithm, which measures current store performance to determine the correct amount of batching as workloads change (Section 4.3.2).
4.3.1 Overlapped Request Processing

Having multiple storage requests to send in a batch requires the application to overlap its own processing of incoming client requests. Figure 4.2 illustrates overlapped request processing for both reads and writes. Note that only reads that miss the middle-tier’s cache require a request to the store; cache hits are serviced directly at the middle-tier. Initially, the application receives two client requests, “Change 1 on A” and “Change 2 on B”. Both of the client requests are processed up to the point that they generate requests for the store. These are then sent in a single batch to the store. After the store acknowledgment arrives, replies are sent to both of the client requests. While waiting for the store acknowledgment, client request “Change 3 on B” arrives and is processed up to the point of generating a request to the store. Later, client request “Read B” arrives and hits the middle-tier cache, while “Read C” arrives and requires fetching C from the store. We describe in Section 4.4 how the Stout implementation handles the multiplexing of these storage requests into batches and the corresponding de-multiplexing of store responses. Grouping storage requests together enables two well-known optimizations:

- **Batching**: Many stores perform better when a set of operations is performed as a group, and many systems incorporate a group-commit optimization [23, 59, 62].

*Figure 4.2: An example of overlapped request processing.*
The performance improvements arise from a number of factors, such as reducing the number of commit operations performed on the transaction log, or reducing disk seek time by scheduling disk operations over a larger set. Storage system performance further improves by initiating batching from the middle-tier for reasons that include reduced network and protocol processing overheads.

- **Write collapsing:** When multiple writes quickly occur on the same object, it can be significantly more efficient for the middle-tier server to send only the final object state. An example where write collapsing may arise in cloud services is tracking advertising impressions, where many clients may increment a single counter in quick succession and the number of writes can be safely reduced by writing only the final counter value to the store. Many workloads possess opportunities for write collapsing, and many prior systems are designed to exploit these opportunities [117, 130].

Stout’s novelty is managing how these optimizations are exploited for a shared remote store based on a multiplicative-increase multiplicative-decrease (MIMD) control loop. It does this by varying a single parameter, the batching interval. At the end of each interval, Stout sends all writes and cache-miss reads to the store. In this way, the batch size is simply all such reads and writes generated in the previous interval, and write collapsing is obtained to the extent that multiple updates to the same key happened during this interval. Pipelining occurs if this batch is sent to the store while an earlier batch is still outstanding (i.e., when the batching interval is less than the store latency).

For a given workload, a longer batching interval will allow more requests to accumulate, leading to a larger batch size and potentially greater throughput. Table 4.1 quantifies the improvement in maximum throughput for one of the experimental configurations that we use to evaluate Stout. This configuration is described in detail in Section 4.5.2. Our goal here is simply to convey the magnitude of potential throughput gain (over 50%) from even slightly lengthening the batching interval. This throughput gain translates into a much larger set of workloads that can be satisfied without queues building up at the store and requests eventually being dropped.

However, the improved throughput of a longer batching interval is not always needed; if the workload is sufficiently light, client latency is minimized by sending
Table 4.1: How a service’s maximum throughput can increase by exploiting batching.

<table>
<thead>
<tr>
<th>Batching Interval</th>
<th>No batching</th>
<th>10ms</th>
<th>20ms</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requests/second</td>
<td>11k</td>
<td>13k</td>
<td>17k</td>
</tr>
<tr>
<td>Throughput Gain</td>
<td>-</td>
<td>18%</td>
<td>55%</td>
</tr>
</tbody>
</table>

Every request to the store immediately. For example, the batching intervals that lead to the higher throughput shown in Table 4.1 also add tens of milliseconds to latency. To determine the right batching interval at any given point in time, Stout measures the current performance of the store. Stout uses these measurements to set its batching interval to be shorter if the store is lightly loaded, and longer if the store is heavily loaded.

### 4.3.2 Updating the Batching Interval

The problem of updating the batching interval is a classic congestion control problem: competing requests originate independently from a number of senders (i.e., middle-tiers); these requests have to share a limited resource—the store—and there is some delay before resource oversubscription is noticed by the sender (in this case, the time until the store completes the request). Like TCP, Stout does not require explicit feedback about the degree of store utilization. This allows Stout to be easily deployed with a wide range of existing storage systems. Unlike TCP, Stout must react primarily to delay rather than loss, as stores typically queue extensively before dropping requests. Thus, our design for Stout’s control loop borrows from a recent delay-based TCP, Compound TCP (CTCP [122]). In general, delay-based TCP variants react when the current latency deviates from a baseline, falling back to traditional TCP behavior in the event of packet loss. Compared to TCP Vegas [21] (another delay-based TCP), CTCP more rapidly adjusts its congestion window so that it can better exploit high bandwidth-delay product links. For Stout, rapid adjustment means faster convergence to a good batching interval.

However, one aspect of our problem differs from that addressed by congestion control protocols. Delay-based TCP assumes that increasing delay reflects congestion
and will consequently reduce the sending rate to alleviate that congestion. Stout acts to reduce congestion by improving per-request performance rather than reducing send rates. Increasing the batch size means that the next request will take longer to process even in the absence of congestion. Furthermore, Stout must distinguish this increased delay due to an increased batch size from increased delay due to congestion. For this reason, Stout has to depart from CTCP by incorporating throughput, not just delay, into measuring current store performance and assessing whether the store is congested.

The remainder of this section describes Stout’s approach to updating the batching interval, which we denote by $\text{intrvl}$, the time in milliseconds between sending batches of requests to the store. In Section 4.3.2, we describe how Stout decides when it is time to update the batching interval. In Section 4.3.2, we describe how Stout decides whether to increase or decrease the batching interval. Increasing the batching interval corresponds to backing off—going slower because of the threat of congestion—while decreasing the batching interval corresponds to accelerating. Then in Section 4.3.2, we describe how Stout decides how much to increase or decrease the batching interval.

**When to Back-off or Accelerate**

Like TCP and its many variants, Stout is self clocking: it decides whether or not to back-off more frequently when the store is fast, and less frequently when the store is slow. To this end, Stout tracks the latency between when it sends a request to the store and when it receives a response. Stout computes the mean of these latencies over every request that completes since the last decision to adjust $\text{intrvl}$; we abbreviate the mean latency as $\text{lat}$.

Stout decides to either back-off or accelerate as soon as both $\text{MinRequests}$ re-
quests have completed and \((\text{MinLatencyFrac} \times \text{lat})\) time has elapsed; the former term is dominant when there is little pipelining, and the latter term is dominant when there is significant request pipelining. We find that this waiting policy mitigates much of the jitter in latency measurements across individual store operations. Table 4.2 shows the settings for these parameters that we used in our experiments, as well as the other parameters (introduced later in this section) that play a role in making Stout robust to jitter.

**Whether to Back-off or Accelerate**

Stout makes its decision on whether to back-off or accelerate by comparing the current performance of the store to the performance of the store in the recent past. We denote the store’s current performance by \(\text{perf}\), its recent performance by \(\text{perf}^*\), and we explain how both are calculated over the next several paragraphs. As mentioned in the Introduction, Stout restricts its measurements to response times so that it can be re-used on different stores, as this measurement requires no store-specific support. The performance comparison is done with some slack (denoted as \(\text{thresh}\)), so as to avoid sensitivity to small amounts of jitter in the measurements:

\[
\text{if } (\text{perf} < (\text{thresh} \times \text{perf}^*)) \\
\text{BACK-OFF} \\
\text{else} \\
\text{ACCELERATE}
\]

We calculate \(\text{perf}\) using the number of bytes sent to and received from the store during the most recent self-clocking window (denoted by \(\text{bytes}\)), the mean latency of operations that completed during this same period of time, and the length of the current batching interval. (Note that higher \(\text{perf}\) is better.)

\[
\text{perf} = \frac{\text{bytes}}{\text{lat} + \text{intrvl}}
\]

Our \(\text{perf}\) definition is a simple combination of latency and throughput: Stout’s latency is \(\text{intrvl} + \text{lat}\), the time until Stout initiates a batch plus the time until the store responds; Stout’s throughput is \(\text{bytes/intrvl}\), the amount sent divided by how often it is sent.
Incorporating throughput rewards backing-off when it causes throughput to increase and the throughput improvement outweighs the larger store latency (lat) from processing a larger batch. By contrast, just measuring latency could lead to an undesirable feedback loop: Stout could back-off (taking more time between batches), each batch could send more work and hence take longer, the store would appear to be performing worse, and Stout could back-off again.

Stout must compute recent performance (perf*) in a manner that is robust to background noise, is sensitive to the effects of Stout’s own decisions, and that copes with delay between its changes and the measurement of those changes. To this end, Stout computes perf* over different sets of recent measurements depending on its own recent actions (e.g., backing off or accelerating). To explain the perf* computation, we first present the algorithm and then provide its justification.

\[
\text{if (last decision was ACCELERATE)} \hspace{1cm} \text{perf}^* = \text{MAX}_i \left( \frac{\text{bytes}_i}{\text{lat}_i + \text{intrvl}_i} \right) \quad (1)
\]

\[
\text{else} // \text{last decision was BACK-OFF} \hspace{1cm} \text{if (intrvl < EWMA(intrvl_i))} \hspace{1cm} \text{perf}^* = \frac{\text{EWMA(bytes}_i)}{\text{EWMA(lat}_i) + \text{EWMA(intrvl}_i)} \quad (2)
\]

\[
\text{else} // (\text{intrvl} \geq \text{EWMA(intrvl}_i)) \hspace{1cm} \text{perf}^* = \frac{\text{EWMA(bytes}_i)}{\text{EWMA(lat}_i) + \text{intrvl}} \quad (3)
\]

Equation (2) for computing perf* is the most straightforward: it is an exponentially weighted moving average (EWMA) over all intervals i since the last acceleration. However, Stout cannot always wait for latency changes to be reflected in this EWMA because of the risk of overshooting—not reacting quickly enough to latency changes that Stout itself is causing. This risk motivates Equations (1) and (3), which we now discuss.

Equation (1) prevents overshoot while accelerating. When Stout is accelerating, it runs a risk of causing the store to start queuing. To prevent this, Stout heightens its sensitivity to the onset of queuing by computing recent performance (perf*) as the best performance since the last time Stout backed-off. Stout stops accelerating as soon as current performance drops behind this best performance. By contrast, calculating recent performance using an EWMA would mask any latency increase due to queuing until it
had been incorporated into the EWMA multiple times.

Equation (3) prevents overshoot while backing-off: when Stout backs off, the increase in \( \text{intrvl} \) can penalize current \( \text{perf} \), potentially causing Stout to back-off yet again, even if throughput (the \( \text{bytes/lat} \) portion of \( \text{perf} \)) has improved. To address this, when the current \( \text{intrvl} \) is larger than its recent history, we use it in calculating both \( \text{perf} \) and \( \text{perf}^* \).

**How Much to Back-off or Accelerate**

Stout reuses the MIMD-variant from CTCP [122]: MIMD allows ramping up and down quickly, and as in CTCP, incorporating \( \sqrt{\text{intrvl}} \) into the update rule provides fairness between competing clients. A minor difference between CTCP and Stout is that CTCP modifies the TCP window, and backing-off corresponds to decreasing this window; Stout modifies its batch interval, and backing-off corresponds to increasing this interval.

Stout backs off using a simple multiplicative back-off step, and it accelerates using a multiplicative factor that decreases as \( \text{intrvl} \) approaches its lower limit (1 ms in this case):

**BACK-OFF:**

\[
\text{intrvl}_{i+1} = (1 + \alpha) \times \text{intrvl}_i
\]

**ACCELERATE:**

\[
\text{intrvl}_{i+1} = (1 - \beta) \times \text{intrvl}_i + \beta \times \sqrt{\text{intrvl}_i}
\]

Competing clients converge to fairness because slow clients accelerate more than fast clients when the store is free, and all clients back-off by an equal factor when the store is busy. The CTCP paper formally analyzes this convergence behavior [122].

The incremental benefit of additional batching decreases as the batch size grows. Because of this, Stout must react more dramatically if the store is already processing large batches and then starts to queue. To accomplish this, we make \( \alpha \) (the back-off factor) proportional to an EWMA of latency, with an upper bound:

\[
\alpha = \text{MAX}(\text{EWMA}(\text{lat}_i) \times \alpha', \alpha_{\text{max}})
\]

Finally, stores occasionally exhibit brief pauses in processing, leading to short-lived latency spikes (this behavior is described in greater detail in Section 4.5.7). This
Table 4.3: Parameters for gain and boundary conditions. These parameters are analogous to those in CTCP, e.g., $\text{intrvl}_{\text{max}}$ corresponds to $RTO_{\text{max}}$.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\alpha'$</td>
<td>1/400</td>
</tr>
<tr>
<td>$\alpha_{\text{max}}$</td>
<td>1/2</td>
</tr>
<tr>
<td>$\beta$</td>
<td>1/10</td>
</tr>
<tr>
<td>$\text{intrvl}_{\text{initial}}$</td>
<td>80 ms</td>
</tr>
<tr>
<td>$\text{intrvl}_{\text{max}}$</td>
<td>400 ms</td>
</tr>
</tbody>
</table>

behavior could cause Stout to back-off dramatically, and then take a long time recovering. To address this, we introduce an $\text{intrvl}_{\text{max}}$ parameter; just as TCP will never assume that the network has gotten so slow that retransmissions should wait longer than $RTO_{\text{max}}$, Stout will never assume that store performance has degraded to the point that batches should wait longer than $\text{intrvl}_{\text{max}}$. This bounds Stout’s operating range, but allows it to recover much more quickly from brief store pauses.

Table 4.3 shows the gain and boundary condition parameter settings. As in CTCP’s parameter settings, the initial batching interval is conservative, and the gain parameters lead to bigger back-offs than accelerations, similar to how TCP backs off faster than it accelerates. Our experiments in Section 4.5 show that Stout works well with these choices, and that it effectively converges to batching intervals spanning over two orders of magnitude.

4.4 Implementation

Stout’s primary novelty is its algorithm for dynamically adjusting the batching of storage requests. We implement the Stout prototype to evaluate this algorithm with a real-world cloud service (a component of Microsoft’s Live Mesh service [87]). We first describe how the application ensures that each key is owned by a single middle-tier (Section 4.4.1). We then describe the Stout internal architecture (Section 4.4.2), followed by how Stout multiplexes storage requests into batches and the corresponding de-multiplexing of store responses (Section 4.4.3). Finally, we describe the Stout API by walking through an example of its use (Section 4.4.4).
4.4.1 Key Ownership

As discussed previously, applications that use Stout must ensure that all requests on a given partition key are handled by only one middle-tier server at any given point in time. In particular, the write collapsing optimization requires that all updates to a given partition key are being sent to the same server. This requirement could be met using a variety of techniques; the applications we evaluate rely on Centrifuge [3].

Centrifuge is a system that combines lease-management with partitioning. Centrifuge uses a logically centralized manager to divide up a flat namespace of keys across the middle-tier servers. Centrifuge grants leases to the middle-tiers to ensure that responsibility for individual objects within the namespace are assigned to only one server at any given point in time. Front-end Web servers route requests to middle-tiers via Centrifuge’s lookup mechanism.
Table 4.4: Client API. All calls are asynchronous.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stout.Fill(key)</td>
<td>Ask Stout to fetch objects associated with partition key from store.</td>
</tr>
<tr>
<td>Stout.MarkDirty(key)</td>
<td>Mark objects associated with partition key as modified, so that Stout knows to persist them.</td>
</tr>
<tr>
<td>Stout.MarkDeleted(key)</td>
<td>Mark objects associated with partition key as deleted, so that Stout knows to delete them from store.</td>
</tr>
<tr>
<td>Stout.SendMessageWhenSafe(key, sendMsgCallback)</td>
<td>Sends a reply message after Stout’s internal dependency map indicates it is safe to send response.</td>
</tr>
<tr>
<td>Stout.SerializeDone(key[], byte[][][])</td>
<td>App indicates completion of Stout’s request to serialize objects.</td>
</tr>
<tr>
<td>App.Serialize(key[])</td>
<td>Callback invoked by Stout for objects that have been marked dirty. Requests App to convert objects into byte arrays to send to the store and respond with SerializeDone().</td>
</tr>
<tr>
<td>App.Deserialize(key[], byte[][][])</td>
<td>Callback invoked by Stout when Fill() responses arrive from store. Converts each byte[] into object.</td>
</tr>
</tbody>
</table>

4.4.2 Stout Internal Architecture

Stout’s internal architecture divides the problem of managing interaction with the store into three parts, as depicted in Figure 4.3. The “Persistence and Dependency Manager” component handles correctness and ordering constraints (e.g., ensuring that requests are committed to the store before replies are sent), as described in Section 4.4.3. Applications interact with this component through the API described in Section 4.4.4. The “Update Batching Interval” component implements the adaptive batching algorithm from Section 4.3.2. The “Storage Proxy” component is a thin layer that connects Stout to a specific scalable storage system. We have implemented three proxies to interface Stout with different cloud storage systems, and all three use TCP as a transport layer.

4.4.3 Persistence and Dependencies

Each middle-tier uses Stout to manage its in-memory data as a coherent cache of the store. Stout is responsible for communicating with the store and ensuring proper message ordering. The application is then responsible for calling Stout when it: (1) needs to fetch data from the store, (2) modifies data associated with a partition key, or
We have received an 'update' for the spreadsheet named by 'key'.

```
ProcessRequest(update, key) {
  // We ask Stout to get state for this key from the back-end store if not held locally.
  if (!table[key]) Stout.Fill(key);
  ... // Block until Stout fills key

  // Spreadsheet-specific logic is in UpdateSheet().
  replyMsg = UpdateSheet(table[key], update);

  // Tell Stout about update.
  Stout.MarkDirty(key);

  // Ask Stout to send reply when update is persisted.
  Stout.SendMessageWhenSafe(key, replyMsg);
}
```

(a) Placement of API calls in sample application code. Stout and the application communicate via message passing, so the application does not need to coordinate its locking with Stout.

(b) Flow of calls between spreadsheet application, Stout and store. The portion of time when the spreadsheet application is active is denoted by the thick black line.

**Figure 4.5:** An example use case of a spreadsheet application interacting with Stout.

(3) wants to send a reply to a client.

Stout ensures proper message ordering by maintaining a dependency map that consists of two tables, as depicted in Figure 4.4. Keys are added to the table of dirty keys whenever the application notifies Stout that a key has been modified. Messages provided by the application are added to the table of in-progress operations if the key is dirty or there are any outstanding operations to the store on this key; otherwise, the messages are sent out immediately. When Stout sends a batch of writes to the store to commit the new values of some keys, those same keys are removed from the table of dirty keys, and Stout fills in the “Store Op” for the appropriate rows in the table of
in-progress operations. When a store operation returns, Stout sends out messages in the order they were received from the application.

Figure 4.4 depicts both batching (keys 11 and 51 were both sent in storage operation 29) and write collapsing (two update operations for key 11 were both conveyed in operation 29). Stout requires the store to commit operations in order, but the store may still return acknowledgments out of order. In our example, if the acknowledgment of 30 arrives before the acknowledgment of 29, Stout would mark the fourth row of the table “Ready” and send the message once all earlier store operations on key 11 are ready and their messages sent.

### 4.4.4 Stout API

Table 4.4 describes each of the API calls and the callbacks that applications must provide for Stout. Figure 4.5a shows how a data center spreadsheet application places the API calls in its code. Before the application’s `ProcessRequest()` function is called, the application has already received the request, done any necessary authentication, and checked that it holds the lease for the given partition key. `ProcessRequest()` handles both modifying spreadsheet objects (done in `UpdateSheet()`) and interacting with Stout: using Stout to fetch state from the store, letting Stout know that the state has been updated, and telling Stout about a reply that should be sent once the update has been persisted to the store. We do not show the code to send the reply, but note that before the application sends the reply message to the client, it must check that the lease for the partition key has been continuously held for the duration of the operation.

Figure 4.5b illustrates the ordering of calls between the application and Stout, and between Stout and the store. When an application or service first receives a request on a given partition key, it fetches the state associated with that partition key using the `Stout.Fill()` call. When the state arrives, Stout calls `App.Deserialize()` to create in-memory versions of fetched objects, which can then easily be operated on by the application logic.

To support coherence, Stout needs to know when operations modify internal service state, so that these updates can be saved to the store. Since Stout has no a priori knowledge of the application internals, Stout requires the service developer to call
Stout.MarkDirty() in any service methods that modify objects associated with a partition key. At some point after a key has been marked as dirty, the Stout persistence manager will call App.Serialize() on a set of dirty keys. By delaying calls to App.Serialize(), Stout allows modifications to the same object to overwrite each other in-memory, thus capturing write collapsing. The application then responds by calling Stout.SerializeDone() with the corresponding byte arrays to be sent to the store.

When a Stout-enabled service would like to send a response to a user’s request, it must use Stout.SendMessageWhenSafe() to provide the outgoing message callback to Stout. Stout will then take responsibility for determining when it is safe to send the outgoing message, based on its knowledge of the current interactions with the persistent store related to the partition key for that request. For example, if the message is dependent on state which has not yet been committed to the persistent store, Stout cannot release the message until it receives a store acknowledgment that the commit was successful.

For certain services, the state associated with a partition key may be large enough that one does not want to serialize the entire object every time it is modified, especially if the size of the modifications is small compared to the size of the entire state. To handle this case, the API supports an additional parameter, a sub-key. Stout keeps track of the set of dirty sub-keys associated with each partition key, and asks the application for only the byte arrays corresponding to these sub-keys. Finally, Stout also enables deletion from the persistent store using the Stout.MarkDeleted() call, which similarly takes both partition keys and sub-keys. Stout tracks these requested deletes, and then includes them in the next batch sent to the store, along with any read and write operations.

4.5 Evaluation

We now demonstrate the benefits of Stout’s adaptation strategy. In Section 4.5.1, we describe the setup for our experiments. In Section 4.5.2, we evaluate the potential benefits of batching and write collapsing in the absence of adaptation. In Sections 4.5.3-4.5.6, we evaluate Stout’s adaptation strategy and show that it outperforms fixed strategies with both constant and changing workloads, that multiple instances of Stout
dynamically converge to fairly sharing a common store, and that Stout’s adaptation algorithm works across three different cloud storage systems. Finally, in Section 4.5.7, we examine the behavior of our store, and we show that Stout is robust to brief “hiccups” where the store stops processing requests.

4.5.1 Experimental Setup

We first describe the application that we ported to use Stout and this application’s workload, and we then characterize the system configuration for our experiments.

Application and Workload

The application we run on our middle-tier servers is a “sectioned document” service. This service is currently in production use, and additional details can be found in the Centrifuge paper [3]. This service allows documents to contain independent sections that can be named, queried, added, and removed. The unmodified service is approximately 7k commented lines of C# code, and we ported this service to use the Stout API changing approximately 300 lines of code. Stout itself consists of 4k commented lines of code and the storage proxies are each approximately 600 commented lines of code.

In production, this service is deployed on multiple large pools of machines. One pool is used exclusively to store device presence: a small amount of addressing information, such as IP address, and an indication whether the device is online. Although we were unable to obtain a trace from production, we used known characteristics of the production system to guide the design of a synthetic client workload for our evaluation: varying request rates on a large number of small documents, 2k documents per middle-tier, each consisting of a single 256-byte section. At saturation, our store is limited by the total number of operations rather than the total number of bytes being stored under this workload, a common situation [27, 93].

In this synthetic workload, we designed the read/write mixture to best evaluate Stout’s ability to adapt under workload variation. We avoid making the workload dominated by reads, because this would have primarily loaded the middle-tiers, and Stout’s goal is to appropriately adapt when the store is highly loaded. We also avoided a pure-write workload because this would not capture how reads that hit the middle-tier cache
are delayed if they touch documents that have been updated but where the update has not yet been committed to the store. This led us to choose a balanced request mixture of 50% reads and 50% writes.

In the commercial cloud service that motivates our workload, all data fits in RAM—Stout is using the store for persistence, not capacity. Because of this, read latencies are uniformly lower than write latencies (e.g., Figure 4.9 in Section 4.5.3). In the Stout consistency model, write latencies impact the user experience because responses are only sent after persisting state changes (e.g., after saving a spreadsheet update). Because writes form the half of the workload that poses the greater risk of poor responsiveness, the rest of the evaluation reports only write latencies unless otherwise noted.

**System Configuration**

Our testbed consists of 50 machines with dual-socket quad-core Intel Xeon 5420 CPUs clocked at 2.5 GHz, with 16 GB of RAM and \(2 \times 1 \text{TB SATA 7200 rpm drives}\). We chose the ratio of front-ends to middle-tiers to storage nodes such that the overall system throughput was maximized subject to the constraint that the storage system was the bottleneck. This led to dividing the 50 machines into 1 experiment controller, 1 Centrifuge lease manager, 12 front-ends that also generate the synthetic client workload, 32 middle-tiers using the Stout library, and 4 systems running the persistent storage system. The choice of 32 middle-tiers means there are 64k total documents in the system. Unless noted otherwise, latency is measured from the front-ends (denoted FE latency in the figures)—this represents the part of end-to-end client latency due to the data center application.

Most of our experiments run Microsoft SQL Server 2008 Express on each of the four storage servers to implement persistent storage. We configure the storage servers to use a dedicated disk for SQL logging, and we followed the SQL documentation to ensure persistence under power loss, including disabling write-caching on our SATA drives [91]. The Stout storage proxy consists of a simple client library that performs hash-based partitioning of the database namespace. For a small number of experiments, we used two additional stores: the PacificA storage system [80] which uses log-based storage and replication, and the commercially available SQL Data Services (SDS) cloud-
**Figure 4.6:** Two fixed batching intervals (10 ms, 20 ms) on a workload with low write collapsing (10k documents) or high write collapsing (100 documents).

Based storage system [88]. Under our workload, these stores occasionally exhibit brief hiccups where they pause in processing; we describe this in more detail in Section 4.5.7. Unless noted otherwise, we report data from runs without hiccups.

### 4.5.2 Batching and Write Collapsing

We perform two experiments to evaluate the potential performance improvements that are enabled by the batching and write collapsing optimizations. For both experiments, we use two different fixed batching intervals—10 and 20 ms—to isolate the benefits of batching and write collapsing from adaptation.

Figure 4.6 shows the performance benefits of write collapsing. For this experiment, requests are delayed for the duration of the batching interval, but they are not actually sent in a batch; at the end of each batching interval, all the accumulated requests are sent individually to the store. Because of this, the entire observed performance difference is due to write collapsing. The low collapsing workload consists of 10k documents
spread across the 32 middle-tiers, while high collapsing consists of only 100 documents, significantly increasing the probability that there are multiple updates to the same document within the batching interval. The graph shows that, as expected, write collapsing reduces latency and improves the capacity of the system. For the low collapsing case, we see that the 10- and 20-ms batching intervals can satisfy between 4k requests/second and 10k requests/second with better client perceived latency for a 10-ms batching interval. However, at 12k requests/second the storage system is overloaded, resulting in a large queuing delay represented by an almost vertical line. In contrast, for the high collapsing workload a 10-ms batching interval can sustain nearly 15k requests/second because the actual number of writes sent to the store is reduced. For the 20-ms batching interval, the number of writes is reduced enough to shift the bottleneck from the store to the middle-tier and provide up to 80k requests/second.

Figure 4.7 shows the performance benefits of batching. The no-batching experiments reflect disabling batching using the same methodology as in the write collapsing experiment: requests are delayed but then sent individually. We see that the throughput
benefits of batching are noticeable at 10 ms, and they increase as the batching interval gets longer, which in turn causes the batch size to get larger. At a 20-ms batching interval, batching allows the system to handle an additional 6k requests per second. The amount of write collapsing for each fixed batching interval in this experiment is constant (and small). We separately observed that PacificA also delivers performance benefits from batching (this is detailed in Section 4.5.6, where we evaluate Stout on both PacificA and SDS). As mentioned in the Introduction, the reason for batching’s benefits depend on the individual store being used; for our partitioned store built on SQL, we separately determined that a significant portion of the benefit comes from submitting many updates as part of a single transaction.

4.5.3 Adaptive vs. Fixed Batching

In this section, we demonstrate that Stout is effective across a wide operating range of offered loads, and investigate the overhead imposed by Stout’s adaptation over
Figure 4.9: Mean response latency for reads: Stout versus fixed batching intervals over a wide variety of loads. Note that the y-axis is $4 \times$ smaller than in Figure 4.8.

Figures 4.8 and 4.9 compare Stout to fixed batching intervals that vary from 20 ms up to 160 ms, for offered loads that range from 5k requests/second all the way up to 41k requests/second, which is very near the maximum load that our storage system can support. These figures are generated from the same experiments: Figure 4.8 shows the mean response latency for write operations whereas Figure 4.9 shows the latency for reads – all reads are cache hits in this workload, but the latency numbers do include delay from reading an updated document where the update has not yet been committed to the store. In both graphs, we see that Stout provides a wider operating range than any of the fixed batching intervals, and it provides response latencies that are either similar to or better than the fixed batching intervals. Looking at the two extremes of latency and throughput in Figure 4.8, Stout’s 4.2 ms latency at 6k requests/second is over $34 \times$ smaller than the 144 ms latency incurred by the longest fixed batching interval in this experiment (160 ms), while Stout’s 41k requests/second maximum is over $3 \times$ larger than the 12k requests/second maximum for the shortest fixed batching interval in this
Figure 4.10: Latency of responses for Stout (a, b) and fixed batching intervals (c, d), at two different workloads, 24k requests/second (a, c) and 26.4k requests/second (b, d). In (a, b), we see Stout’s changing response latency overlayed with its mean response latency. In (c, d), Stout’s mean response latency is overlayed with the mean latency and standard deviation for multiple fixed batching intervals. The slight increase in request/s/second causes the best fixed interval from 24k requests/second to generate queuing at 26.4k requests/second.

To understand the overhead of Stout’s adaptation, we compare Stout to different fixed batching intervals at fine granularity under two fixed workloads. In Figures 4.10 (a) and (b), the time series show Stout’s latency to be relatively steady, and for this reason we focus on the mean latency throughout this section. Figure 4.10 (c) compares Stout’s mean to fixed intervals with an offered load of 24k requests/second. The best fixed interval is at 50 ms, and here we observe that Stout’s adaptation adds just under 15 ms to the response latency (from 80 to 94 ms) and is within the standard deviation. When the fixed batching interval is too short (40 ms), the store is overloaded and we see large queuing delays. When the fixed interval is too long (at 70 ms and above), we see unnecessary latency. Figure 4.10 (d) shows a similar comparison, but with an offered load of 26.4k requests/second. Here we see that the best fixed interval is at 60 ms, and the overhead imposed by Stout’s adaptation is about 25 ms (from 75 to 100 ms), again within the standard deviation. If we use the best fixed interval from 24k requests/second (50 ms), the store becomes overloaded and unable to process requests in a timely fashion.
Figure 4.11: Stout outperforms a fixed batching interval after the load either increases or decreases.

until the load subsides. These results demonstrate the need for adaptation—choosing the right fixed interval is difficult, even with this modest difference in offered load.

4.5.4 Dynamic Load Changes

Thus far we have shown Stout operating over fixed request rates. Here, we explore Stout’s response to a sudden change in request load. For this experiment we apply a fixed load of 12k requests/second to our standard configuration and part way through the experiment we change the request load. Figure 4.11 shows the front-end latency for two of these experiments. In the first experiment, the load decreases to 6k requests/second. The front-end latency for Stout decreases from 50 ms to 5 ms. In the second experiment, the load increases to 18k requests/second and the latency increases from 50 ms to 80 ms. In contrast, a 20-ms fixed interval is marginally better than Stout at 12k requests/second but it only achieves 24 ms after the decrease and it causes queuing at the store after the increase. This demonstrates Stout’s benefits in the presence of workload
changes.

4.5.5 Fairness

Cloud storage systems typically serve many middle-tiers and it is important that these middle-tiers obtain fair usage of the store. To measure Stout’s ability to converge to fairness, we ran an experiment where after 90 seconds, we forcibly set half of the thirty-two middle-tiers to a batching interval of 400 ms and the remaining half to 80 ms. The middle-tier servers then collectively reconverge to the steady state. Because Centrifuge balances the distribution of documents across the middle-tiers, they have identical throughput throughout the experiment and we are only concerned with latency-fairness. The middle-tiers achieve good fairness after re-convergence: measuring from 30 seconds after the perturbation to 120 seconds after the perturbation, the mean latencies have a Jain’s Fairness [69] of 0.97, where a value of 1.0 is optimal.

4.5.6 Alternate Storage Layers

To explore the generality of Stout’s adaptation algorithm, we run experiments using two additional storage platforms with substantially different architectures. For both, we keep the same algorithm but calibrate the parameters to the new store. We first evaluate Stout against SQL Data Services (SDS) [88], a pre-release commercial storage system. For SDS, we calibrate the parameters to be the same as in Section 4.3.2 except that $thresh = 0.2$ and $\beta = 1/4$. The current SDS API does not support batching or pipelining, and thus the best approach in our workloads is to send as rapidly as possible. We find that Stout does converge to sending as rapidly as possible.

We also evaluate Stout against PacificA [80], a research system that differs from our SQL-based storage layer in that it includes replication and uses log-based storage. We configure PacificA with three-way partitioning and three-way replication for a total of nine storage machines and one additional metadata server. The rest of the setup consists of twelve front-ends, sixteen middle-tiers, and one Centrifuge manager server. We calibrate the parameters from Section 4.3.2 to have EWMA-factor $= 1/32$, $thresh = 0.7$, and $\beta = 1/8$. Figure 4.12 shows Stout’s behavior across a range of request loads. At
Figure 4.12: Mean response latency for writes using PacificA: Stout and fixed intervals over a variety of loads.

low to moderate load, Stout compares favorably to the best (20- and 40-ms) fixed batching intervals. As load increases, PacificA’s log compaction frequency also increases, resulting in sufficiently frequent store hiccups that we are not able to avoid them in our experiments. After 22.2k requests/second, Stout has difficulty differentiating the store hiccups from the queuing behavior to which it is adapting. In spite of these hiccups, Stout outperforms any fixed batching interval in the presence of significant workload variation: compared to the short intervals, it avoids queuing at high loads; compared to long intervals, it yields much better latency at low loads.

4.5.7 Store Hiccups

As mentioned in our experiments with PacificA, stores sometimes experience hiccups, where they briefly pause in processing new requests. Such Stout-independent hiccups can lead to large spikes in observed latency, complicating Stout’s task of inferring store load. We now investigate the issue of hiccups in more detail.
Figure 4.13 shows the occasional brief pauses in processing (or “hiccups”) that occur over a 2-hour interval when using the SQL Server storage system. For this experiment, we used a single middle-tier server sending 3k operations per second with a fixed 2-ms batching interval to a single SQL Server back-end machine, and we measured latency from within the Stout storage proxy — this is denoted SP latency and it only includes the time to send the requests over a TCP connection to the back-end and the time that the store takes to service these requests and send responses back to the middle-tier. The figure shows that these hiccups occur on an irregular and infrequent basis, and they lead to significant spikes in latency — up to three orders of magnitude greater than the steady state. Although this figure only shows the hiccups at one offered load, we have run similar experiments with different loads, and we have not observed any obvious correlation between the offered load and the frequency of hiccups in this store.

Although we do not know the exact cause of hiccups in the SQL store, we believe they are caused by periodic background bookkeeping tasks that are common in storage systems. We did make efforts to eliminate such hiccups from SQL Server by both disabling the option that generates query-planning statistics and setting the recovery interval to one hour (the recovery interval controls how much replay from the log may be needed after a crash). These changes reduced the number of hiccups but did not eliminate them. As mentioned in Section 4.5.6, we observed that log compaction is responsible for even more frequent hiccups in PacificA.

Because these brief latency spikes may be unrelated to the offered load, an appropriate response to them is simply to pause briefly; increasing the batching interval is not appropriate because the store is not actually overloaded. The problem of a unrelated event causing the appearance of congestion is familiar from the literature on TCP over wireless channels, where packet loss may reflect either congestion (which should be mitigated by the sender) or background channel noise (which can frequently be ignored). In response, researchers have proposed explicit signaling techniques like ECN [12, 76] to improve performance in these challenging environments. Our measurements suggest that similar mechanisms for adaptive use of cloud storage are also worth researching. In this chapter, we restrict our attention to showing that Stout, which does not try to distin-
Figure 4.13: Intermittent hiccups in store processing yield brief spikes in latency as measured from the middle tier. These measurements were taken with a 2-ms fixed batching interval and 3k requests/second.

guise latency due to store hiccups from latency due to overload, still copes acceptably with such hiccups.

Figure 4.14 shows how Stout reacts to one of these hiccups: the solid line shows the measured response time of the store, and the dashed line shows how Stout adjusts its intrvl as a result of the latency spike. With intrvl_max set to 400 ms, Stout takes slightly over half a minute to recover from the very large spike in latency (the peak in this figure is 2,696 ms) caused by this hiccup. This recovery is rapid compared to the frequency of hiccups. Lowering intrvl_max would improve recovery time, but would also reduce Stout’s operating range.

The rarity of store hiccups raises a methodological question: each of our experiments would have needed to run for hours in order for the number of hiccups to be similar across runs. Because Figure 4.8 alone includes 27 such experiments, such an approach would have significantly hindered our ability to evaluate Stout under a wide variety of conditions. Because Stout recovers from store hiccups with reasonable speed,
we chose instead to re-run the occasional experiment that saw such a hiccup. The one exception is our experiment using PacificA (Section 4.5.6), where hiccups were sufficiently frequent that we did not need to take any special steps to ensure a comparable number across runs.

### 4.6 Related work

Stout’s control loop is inspired by the literature on TCP and, more generally, adaptive control in computer systems. The Stout implementation also incorporates a number of well-known techniques from storage systems. We briefly discuss a representative set of this related work.

There is a large existing literature on TCP [68, 72, 138]. This prior work has explored many different indicators of utilization and load; Stout uses response time measurements to adjust its rate of sending requests to the store. In this regard, Stout is similar to TCP Vegas [21], FAST TCP [135] and Compound TCP (CTCP) [122], each of which attempts to tune the transmit rate of a TCP flow based upon the inter-packet delay.
intervals. In comparison, Stout’s control loop has to deal with the additional subtlety of distinguishing delay due to congestion from delay due to sending a larger batch.

Control theory is a deep field with many applications to computer systems [136, 123, 28, 103, 114, 30]. Despite these successes, many adaptation problems in computer systems have remained unaddressable by control theory due to the dramatic differences between computer systems and the systems that control theory has traditionally considered [66]. For example, advocates of a class of controllers called self-tuning regulators have constructed a list of eight requirements that computer systems must satisfy to enable their successful application [71]. Scale-out storage systems fail to satisfy a number of these conditions, such as the requirement for a modest bound on the actuation delay of the system (e.g., if an application enqueues a large number of requests, future request batching can take a very long time to reduce user-perceived latency). Other control techniques may remove this particular requirement, but instead introduce other difficult requirements, such as the need for a detailed model of scale-out storage system performance [71].

The Stout implementation borrows from prior work on storage systems in two major ways. First, the performance benefits of batching, write collapsing and pipelining are well-known, and have been leveraged by systems such as Lightweight Recoverable Virtual Memory (LRVM) [117], Low-Bandwidth File System [94], Farsite [2], Cedar [59], Practical BFT [23], Tandem’s B30 system [62] and the buffer cache [130]. Stout’s novelty is in using a control loop to manage exploiting these optimizations, not the optimizations themselves.

Second, Stout’s internal architecture incorporates at least two major ideas from prior storage systems. Splitting consistency management from storage was explored in Frangipani [25] and LRVM [117], while prior work such as Soft Updates [50], Generalized File System Dependencies [48], and xsyncfs [99] explored ways to provide some or all of the performance benefits of delayed writes with better consistency guarantees.
4.7 Summary

Stout’s adaptation algorithm is the first technique for automatically adapting application usage of scalable key-value storage systems. Stout treats store access as a congestion control problem, measuring the application-perceived latency and throughput of the store, and dynamically adjusting the application’s grouping of requests to the store. To evaluate this algorithm, we implemented the Stout system and modified a real-world cloud service to use Stout. We found that in the presence of significant workload variation, Stout dramatically outperforms existing non-adaptive approaches with $34 \times$ lower latency compared to a throughput-oriented system and up to $3 \times$ as many requests compared to a latency-oriented system.

Stout’s optimization of storage performance and any other optimizations to the application as a whole are largely complementary to the benefits of scc. The impact of any planned performance improvements can be incorporated into the model to ensure that the hardware can be matched to improved versions of the software. In the next chapter, we move on to consider energy efficiency and the relationship between the execution of software and the power consumption of the hardware.

Chapter 4, in part, is a reprint of material as it appears in the article “Stout: An Adaptive Interface to Scalable Cloud Storage” by John C. McCullough, John Dunagan, Alec Wolman, and Alex C. Snoeren which appears in Proceedings of the USENIX Annual Technical Conference, 2010. The dissertation author was the primary investigator and author of this article.
Chapter 5

Model-Based Power Characterization

We addressed lowering provisioning costs in Chapter 3 and improving storage system performance in Chapter 4. These both influence the server portion of data center costs presented in Chapter 1. The power consumed by the server hardware constitutes the next largest cost component. Managing that power consumption to lower cost, using either existing techniques described in Chapter 2 or future techniques, requires an understanding of the relationship between the application workloads and power consumption.

The most direct way to measure power consumption is to attach power meters to individual machines to understand the impact of the power management decisions. Power meters can be expensive. An alternative is to develop a lightweight power model that can estimate the instantaneous power consumption of a machine and its components. A power model maps statistics collected on a machine to the estimated power consumption. The accuracy of the power estimate then influences the accuracy of the measured effectiveness of the potential power management techniques across the whole cluster or within a specific system.

The relationship between an application and power consumption depends on the particular pieces of hardware in use and how the hardware behaves when idle. Efforts to improve power efficiency have led to multi-core processors and power optimizing technologies, such as clock gating. These improvements make this relationship increasingly complex. Existing efforts to correlate workload-related system properties with power consumption pre-date many of these power efficiency techniques. It is important to ascertain whether the modeling techniques are effective for contemporary systems to
ensure that the accuracy expectations from earlier hardware do not lead to erroneous conclusions.

In this chapter we evaluate the applicability of lightweight power modeling on modern hardware to better understand how the workload-power relationship is influenced by hardware trends. First, we consider how previous power modeling techniques correlate system utilization and hardware metrics with total system power. Next, we show that even more advanced techniques are insufficient for accurate subsystem power modeling. Finally, we discuss why these modeling techniques are likely to grow less accurate in the presence of hardware variability and hidden device states.

5.1 Introduction

Electrical power is a precious resource and its consumption is important to all forms of computing platforms from handheld devices to data centers. Numerous research efforts seek to optimize and carefully manage energy consumption at multiple levels—starting from individual components and subsystems such as wireless radios [105], storage devices [78], and processors [45, 106], to entire platforms [46, 139].

An important goal of these optimizations has been to achieve power proportionality, that is, power consumption that is proportional to the computational work done. As a result, a modern system’s instantaneous power draw can vary dramatically. Moreover, the exact relationship between power draw and activity level is becoming increasingly complex due to the advent of microprocessors with multiple cores and built-in fine-grained thermal control, as well as hidden device states that are not necessarily exposed to the operating system.

While increasingly energy efficient components are a key enabler, achieving the goal of platform-wide power proportionality requires an intelligent dynamic power management (DPM) scheme. A critical first step towards that goal is to characterize how much power is being consumed, both by the platform as a whole and also by individual subsystems. Armed with a reasonable characterization of power consumption, a DPM system then needs to accurately predict how changes in utilization will impact future power consumption. For example, a DPM system can guide resource allocation deci-
sions between heterogeneous but functionally similar resources such as multiple radios on the same platform [105].

Current approaches to prediction rely on assumptions of power proportionality to make architectural or system-level tradeoffs. These tradeoffs are evaluated by building a model of the system that describes power consumption in terms of power states and attributing the model to one or more observable hardware and software counters, correlating these with changes in measured power draw. Previous work on power modeling has focused on modeling total system power consumption using several learning techniques such as linear regression [70, 113], recursive learning automata [83], and stochastic power models [109]. The effectiveness of each of these techniques has been evaluated independently across various benchmarks (see [58] for a review).

However, we are unaware of any definitive comparison of these models for a diverse set of benchmarks on a given platform. Further, the increasingly nuanced relationship between a component’s activity level and power consumption limits the utility of published models in modern systems. In particular, as we shall show, these models are effective only in a restricted set of cases, e.g., when system utilization is constant and very high, for relatively straightforward executions in single cores, or when the system’s static power consumption is dominant and the dynamic component is within the margin of error. Consequently, even well-designed DPM algorithms employing these models will make suboptimal decisions (to shutdown and/or slowdown components) if the actual utilization dynamically changes the significance of various components to overall power consumption.

To overcome the limited ability of models to predict power consumption, manufacturers often build reference systems complete with a large number of sense resistors and use precision analog-to-digital converters (ADC) to create a measurement instrument that accurately captures power consumption at fine time granularities. Research efforts have mimicked this approach by developing custom designs that can breakdown power consumption within sensor platforms [119] or monitor whole-system power for general purpose computers at the power supply [31]. While such extensive direct instrumentation can provide accurate power measurements, it requires significant design effort and increases costs due to board space constraints and the need for additional compo-
nents. Hence, we are not aware of any production systems so instrumented. Moreover, significant increases in cross-part variability [40, 134] limit the applicability of predictions based upon even the measured behavior of a single or small number of reference systems.

We evaluate the need for pervasive power instrumentation by exploring the effectiveness of power modeling on modern hardware. Mindful of the fact that the required level of accuracy varies based upon the specific DPM goal, we consider how well increasingly sophisticated models can predict the power consumption of realistic workloads. We make the following contributions:

- We show empirically that while total system power can be modeled with 1–3% mean relative error across workloads when restricted to a single core, it rises to 2-6% for multi-core benchmarks.

- We find that linear models have significantly higher mean relative error for individual subsystems such as the CPU: 10–14% error on average, but as high as 150% for some workloads. We employ more complex techniques to improve predictive performance, but only by a few percent.

- We present an in-depth analysis of why modeling fails for modern platforms, especially under multi-core workloads. We posit that this poor predictive performance is due to effects such as cache contention, processor performance optimizations, and hidden device states not exposed to the OS. In addition, we present quantitative evidence of significant variability between identical components, which fundamentally bounds the potential accuracy of any modeling based approach.

Taken together these results make a strong case for pervasive instrumentation and dynamic collection of power usage data. While traditionally eschewed due to significant increases in design and manufacturing costs, the advent of relatively inexpensive ADCs and associated circuits makes such an approach increasingly feasible.
5.2 Related work

Researchers have long been interested in optimizing the energy efficiency of computing devices. In the context of mobile platforms, researchers have considered optimizing individual subsystems such as the CPU [45, 106], disk [78] and wireless radios [105]. While much of the early work focused on battery powered devices for usability reasons [46, 105, 139], economic motivations have dramatically increased interest in general purpose computing such as PCs and servers [4, 14, 96].

There have been a number of efforts to predict energy consumption through in-situ power measurements by adding different levels of hardware instrumentation. The Openmoko Freerunner mobile phone platform was designed to support developer access. Carroll and Heiser leveraged its sense-resistor support to characterize power consumption at a component level, deriving simple, time-based linear models for each component [22]. The LEAP platform [119] proposes adding fine-grained instrumentation to all power rails in embedded sensor nodes, and develops the required software instrumentation within the OS kernel to attribute energy to running tasks. In contrast, Quanto [47] proposes a single point of measurement by observing the switching regulator on sensor platforms to increase the practicality of energy attribution. Quanto requires changes to TinyOS and individual applications to track individual state transitions and offline analysis for energy attribution.

These measurement efforts, however, are restricted to special-purpose platforms with limited availability. Moreover, while such detailed, instrumentation-based approaches provide accurate power characterization, they are frequently regarded as too costly to implement at scale. Hence, efforts focused on more general purpose platforms have typically relied upon modeling, mostly for predicting total system power.

Recent activity in server environments is driven by the observation that most computer systems are largely idle [4] or exhibit low utilization except during peak traffic [14]. Hence, Barroso and Hölze argue for more energy-proportional designs so that the energy used is proportional to the utilization [14]. This quest for energy proportionality has led to a variety of low-power server designs [10]. Some applications, however, have been shown to perform poorly on these low-power platforms [112].

Predicting the energy use of a particular application on a general-purpose plat-
form is challenging, however. Previous studies have shown that CPU utilization is not useful by itself and that performance counters used to build models only help if they are correlated with dynamic power [79, 113]. Economou et al. explored the use of component-level measurements including CPU, memory and hard disk but were unable to significantly decrease prediction error [41]. Conversely, Bircher and John explored using performance counters to model not only total system power, but component-level power (e.g., CPU, memory, disk, and I/O) as well in a Pentium IV-class system [18]. They find that linear models are a poor fit for all but the CPU, and instead resort to multiple-input quadratics to provide an average of 9% absolute error.

In the hosted setting, Koller et al. turn to application-level throughput—as opposed to hardware performance counters—to improve power prediction for virtual clusters using linear combinations [75]. The Joulemeter project [70] proposes combining total-system power measurements from server power supplies with power modeling to attribute power consumption at the coarse level of individual virtual machines (VM) running on a physical server. Their model is generated by exercising specific VMs by running particular applications and using the CPU utilization metric to attribute energy.

Despite prediction errors, a number of researchers have demonstrated the utility of power modeling. For example, Bircher and John combine CPU metrics with instantaneous voltage demand information to ensure that the processor uses the correct DVFS setting and achieve an average speedup of 7.3% over Windows Vista’s default DVFS algorithm [19]. Wang and Wang utilize feedback control theory to jointly optimize application-level performance and power consumption [133], while Tolia et al. improve power proportionality by leveraging virtual machines, DVFS, and fan control [127].

5.3 Power characterization

Characterizing the power consumption of a computing platform need not be difficult in principle. Ideally, original equipment manufacturers (OEMs) are well positioned to add extensive power instrumentation to their platforms, which would enable accurate and fine grained power measurements. Combined with such instrumentation, OEMs could further expose interfaces to an operating system to query detailed power
information in a low-overhead manner. This information can then be used by the OS as well as individual applications to manage their energy consumption dynamically. Unfortunately, this ideal scenario is not realized in practice due to manufacturing constraints such as increased board area, cost of components and design costs. Modern platforms are already extremely complex and OEMs are reluctant to add functionality without clear and quantifiable benefits. Hence, while OEMs may have extensive power instrumentation on their *development* platforms during design and testing, we are unaware of any commodity platform that provides fine-grained power measurement capabilities in hardware.

Instead, in the absence of direct power measurement, the commonly used alternative is to make power models. The basic idea behind power modeling is to take as input various software and hardware counters and use those to predict power consumption after suitable training on an appropriately instrumented platform. Regardless, any power characterization approach, whether based upon modeling or direct instrumentation, must trade off between several design alternatives as discussed below.

### 5.3.1 Measurement granularity

One of the dimensions that affects both forms of power characterization—power measurement and power modeling—is the granularity of measurement. Power can be characterized at the level of an entire system (a single power value) or can be done at a logical subsystem granularity, such as the display, CPU, memory and storage subsystems. The appropriate measurement granularity depends on the application. For example, in data centers an application for macro-scale workload consolidation on servers will likely only require total system power measurements at an individual server level. On the other hand, fine-grained scheduling decisions on individual heterogeneous processor cores requires power consumption data for individual cores.

While total system power can be measured at the wall socket directly using myriad commercial devices (e.g. *WattsUP* meters) the applicability is limited in the case of any fine grained adaptation. Furthermore, power measurements at the system level cannot distinguish between the actual power used and the power wasted due to the inefficiencies in the power supply. On the other hand, fine-grained subsystem level power
characterization is more useful since the total system power can still be estimated accurately by adding the power consumption of individual components. Most of the research to date has focused on total system power modeling [41, 113]. In this paper we explore the design space of power characterization and especially investigate the feasibility of accurate power modeling at subsystem granularity.

5.3.2 Accuracy

In the case of power instrumentation, it is possible to get very high levels of accuracy with precision analog-to-digital converters (ADCs), albeit at higher costs. The accuracy of power modeling with respect to the ground-truth measurements is important since the accuracy depends on how well the model fits. Similar to the measurement granularity dimension, the accuracy requirements of power modeling are also somewhat dependent on the application. When consolidating workload to fewer servers in a data center, modeling total power within 5–10% error is sufficient to guide policy decisions since the base power of servers is high [41, 113]. On the other hand, for an application of fine grained scheduling on different heterogeneous processor cores, the accuracy of power characterization needs to be at least as good as the differences in power consumption between the cores, otherwise scheduling decisions may be incorrect.

Furthermore, the required accuracy is likely to vary with particular subsystems based on factors such as their dynamic range, and their contribution to the total system power. Subsystems that are dominant in particular platforms need to be measured more accurately since the penalty of mis-predicting the power is higher. Furthermore, subsystems that are more complex and dynamic, such as processors, need higher accuracy measurements. On our test systems, the CPU consumes between 0.5W and 27W (constituting up to 40% of the total system power), and prediction errors translate to high absolute error. In contrast, an SSD disk drive on the SATA interface, or the network interface, consumes lower power (fewer than 2–3W) and is less dynamic; therefore, higher modeling errors can be tolerated. However, in cases of platforms with a larger number of disks this modeling error will have a more significant impact.
5.3.3 Overhead and complexity

Both power modeling and power instrumentation have associated overheads and complexity. In the case of power instrumentation, OEMs have to integrate the power measurement functionality into their platforms, usually in the form of a shunt resistor connected to ADCs. The ADCs measure the voltage drop across the shunts, which is converted into current and power consumption. Since modern platforms have multiple voltage domains, and subsystems can be powered using +3.3V, +5V, and +12V power supplies, a large number of ADC inputs are required. Furthermore, in case the voltage to the subsystem is not constant, such as with processors that employ dynamic voltage scaling, we need additional ADC inputs to measure voltage as well. Some subsystems, such as processors, can also have multiple power lines powering different functional units within them which each need to be measured separately. The shunt resistors themselves need to be chosen while keeping in mind the dynamic range of the power consumed by individual subsystems, possibly from several milliwatts to tens of watts, to give high precision and low power loss in the sense resistor itself. All of these factors contribute to higher costs—of components, board area and design, test and validation time.

Complexity in power modeling arises in part from the need to capture all the relevant features expressed by software and hardware counters to serve as inputs to build the models. Often platforms and components, such as the CPU, either support tracking a limited set of platform counters simultaneously, or have non-trivial overhead in collecting a large set of counters at fine granularities. Therefore, it is important that the model be sparse and use as few counters and states are possible, while still providing reasonable modeling accuracy. Additionally, the models themselves can be arbitrarily complex and can require non-trivial amounts of computation. In this paper, we explore a series of increasingly complex models to understand the accuracy/complexity tradeoffs.

Finally, in the case of power modeling, transferability or robustness of the power modeling is key: The model should be generated once and should be applicable over time and to other instances of the same platform. While we believe that this is intuitively the case, recent work has highlighted a significant amount of platform heterogeneity, where process and manufacturing variations and aging effects lead to identical hardware exhibiting significant variation in power consumption [24, 40, 134]. We show some
preliminary results relating to this aspect and highlight the associated challenges with power modeling in the face of increasing variability in hardware.

5.4 Power modeling

While there has been a considerable volume of work in the area of power modeling, notably for system power and CPU power, a common thread joining most of the previous work is the assumption that system power can be well predicted by simple linear regression models [41, 70, 113]. Our goal in this paper is to understand whether (i) these simple models are compatible with more contemporary platforms (CPU and platform complexity has increased significantly since many of the previous modeling approaches were proposed), and (ii) whether these models can be applied to individual subsystems within platforms. The latter is important to understand because, with the increasing emphasis on power proportionality and energy awareness, there are several adaptations that can be done at the platform level as well as the subsystem level, provided fine-grained power consumption information is available.

Our initial attempts to use simple linear regression models—including replicating specific ones previously proposed—were disappointing: The models perform poorly on non-trivial workloads. This result could be explained by one of the following reasons:

- The features being fed into the model contain a certain level of cross-dependency, whereas linear models assume feature independence.
- The features used by previously published models are no longer appropriate for contemporary platforms. There may, however, exist a different set of counters that can still lead to a good model.
- Modern hardware components, such as processors, abstract away hardware complexity and do not necessarily expose all the power states to the OS and are thus fundamentally hard to model since changes in power consumption are not necessarily associated with changes in exposed states.
In this section, we describe a number of increasingly complex regression models that we use to fit the power data. Unfortunately, we found that increasing the complexity of the model does not always improve the accuracy of power prediction across the different subsystems that we are trying to model. We begin by describing how linear regression models are constructed and enumerating the specific models we use.

5.4.1 Linear regression models

Let \( y = [y_1, y_2, \ldots, y_k] \) be the vector of power measurements for the \( k \) subsystems and let \( x_i = [x_{i,1}, x_{i,2}, \ldots, x_{i,n}] \) be the normalized vector of measurements on the \( n \) variables (hardware counters and OS variables) collected at time \( t_i \). The linear regression model is expressed as:

\[
y_i = \beta_0 + \beta_1 x_{i,1} + \beta_2 x_{i,2} + \ldots + \beta_n x_{i,n} + \epsilon_i,
\]

where \( \epsilon_i \) is a noise term that can account for measurement error. Thus, the linear regression models are solved by estimating the model parameters \( \beta \), and this is typically done by finding the least squares solution to \( \hat{\beta} = y.X^{-1} \), which can be computed as

\[
\hat{\beta} = \text{argmin}_\beta \sum_{i=1}^{k} \left( y_i - \beta_0 - \sum_{j=1}^{n} \beta_j x_{i,j} \right)^2,
\]

or simply \( \hat{\beta} = \text{argmin}_\beta \left( ||y - \beta X||_2^2 \right) \).

The challenge in building a good power model is to correctly identify the set of \( n \) most relevant features. On the platforms we considered, there are in excess of 800 different hardware counters that can be tracked (even though only a few can be tracked simultaneously). Previous work has overcome this problem by using domain knowledge about the platform architecture to hand pick counters that are believed to be relevant [41, 113]. We believe that increasing complexity in modern processors and platforms makes this task harder with each generation. To understand whether such domain knowledge is critical to power modeling, we also use modeling techniques that perform automatic feature selection in the process of constructing a model. We observe that the features selected by the more complex techniques correspond to the features with the highest mutual information [36] for a given power rail. This makes us confident
that these state-of-the-art modeling techniques are leveraging all relevant features and are not missing anything that is relevant but not linearly correlated.

We now briefly describe the regression techniques that we explore, listed in order of increasing complexity.

**MANTIS:** Ecomomou et al. developed a server power model by fitting a linear regression of four distinct utilization counters obtained from different platform subsystems to the power measurements taken at a wall socket [41]. The input utilization metrics are obtained by running a number of systematic workloads that stress the platform subsystems in sequence.

In particular, they consider counters corresponding to CPU utilization, off-chip memory accesses, and hard drive and network I/O rates. We extend the basic MANTIS linear model to also consider instructions per cycle as a representative baseline obtained from a best-in-class full-system power model [113]. While there have been a large number of efforts focused on identifying a suitable set of performance counters, we choose the MANTIS model as a starting point because it has been shown to have very good predictive properties on previous-generation hardware [41]. Since a few of the counters used by the original MANTIS model are no longer available on modern platforms, we communicated with the authors themselves to find appropriate substitutes.

**Lasso regression:** There are two drawbacks to building a (linear) regression model. First, it requires domain knowledge to identify the correct set of features (possibly from a large space)—in this case, we seek features possibly related to power consumption. Second, when the features are correlated to each other, the regression simply distributes the coefficient weights across the correlated features, and all correlated features are included in the final model, rather than identifying a smaller subset of somewhat independent features. In current complex platforms, there is a very large space of features that can be measured and it is a non-trivial task—even for an expert—to correctly identify the smallest possible subset of power relevant features [118]. Lasso regression, which is a specific instance of $l_1$-regularized regression, overcomes this challenge by penalizing the use of too many features. Thus, it tends to favor the construction of sparse models which incorporate just enough features as are necessary. This is done by incorporating
a penalty factor into the least-squares solution for the regression, which is expressed as:

$$\hat{\beta} = \arg\min_{\beta} \left( \sum_{i=1}^{k} \left( y_i - \beta_0 - \sum_{j=1}^{n} \beta_j x_{i,j} \right)^2 + \lambda \sum_{j=1}^{d} |\beta_j| \right)$$

or, simply $$\hat{\beta} = \arg\min_{\beta} \left( ||y - \beta X||^2_2 + \lambda ||\beta||_2 \right).$$

Here, $$\lambda$$ is a penalty function that encourages the solution to aggressively set $$\beta$$ values to zero (and exclude the associated features from the model). Compared to regular methods, Lasso regression is advantageous since it relies less on strong domain knowledge to pick out the right features; in addition, it is computationally simple, and automatically picks models with a small number of features, which are critical requirements for a usable power model. The optimal value for the $$\lambda$$ parameter is selected by cross validation on the training data. We used the glmnet package to perform the Lasso regression [126].

### 5.4.2 Non-linear regression models

Linear regression models work well when the features being modeled are independent of each other and tend to predict poorly when there are interdependencies between the modeled features; non-linear models can often capture these feature dependencies. (Indeed, previous work has shown that quadratic models can be more effective at modeling subsystem power [18].) The non-linear form of the model can be expressed as:

$$y_i = \beta_0 + \sum_{\ell=1}^{m} \beta_\ell \phi_\ell(x_i) + \epsilon_i,$$

where $$\phi_\ell$$ are non-linear basis functions over the feature vectors $$x_i$$. We use the Lasso least-squares formulation as before to solve the regression and construct a model.

In general, the set of possible $$\phi_j$$ is arbitrarily large and solutions exist for only a few families. We experiment with three well known functions:

**Polynomial with Lasso:** Here, the basis functions are defined as exponentiated forms of the original variables. So, $$\phi = \{x_i^a : 1 \leq a \leq d\}$$ where $$d = 3$$. Again, with Lasso, only the relevant features—now including the polynomial terms which may have cross dependencies—are inserted into the model.
Polynomial + exponential with Lasso: In this slight variation of the previous model, \( \phi \) also includes the functions \( e^{\nu_i} \). As before, we run the full set of terms through the Lasso (linear) regression package which picks out a sparse subset of the terms. In the previous case as well as this one, the optimal \( \lambda \) is selected by cross validation.

Support vector regression (SVR): We also experiment with support vector machine (SVM)-based regression. At a high level, SVMs operate by fitting a hyperplane decision boundary to a set of labeled data instances taken from different classes. The hyperplane boundary is constructed so as to maximize the separation between the two classes of data, and can be used to perform classification, regression, and function estimation tasks on the data. SVMs employ a trick to handle non-linearity, the data is run through a non-linear kernel that maps the data to a higher dimensional space where greater separation may be achieved. An important difference between SVR and Lasso-based methods is that SVR does not force the regression to be sparse. When features are correlated, weights are distributed across them. We use the \texttt{libsvm} \cite{26} and, in particular, the radial basis kernel. The parameters required for the RBF kernel were optimally selected by cross validation on the training data.

5.5 Evaluation setup

We collect platform subsystem power using an instrumented Intel Calpella platform. The platform is a customer reference board that corresponds to the commercially available mobile Calpella platform. This particular board, which is based on the Nehalem processor architecture, was outfitted with an Intel quad-core i7-820QM processor, 2x2GB of DDR3-1033 memory and a SATA SSD drive, running the 2.6.37 Linux kernel. Importantly, we turned off HyperThreading and TurboBoost on the platform to avoid hidden states (these states change operating points but are controlled in hardware and the OS has little visibility into them). A salient feature of this particular board is that it has been extensively instrumented with a very large number of low-tolerance power sense resistors that support direct and accurate power measurements of various subsystems (by connecting to a data acquisition system). The platform contains over one hundred sense resistors and it is a non-trivial task to collect readings from all of them.
Table 5.1: Power characterization for the calpella platform. The 500-W ATX PSU that we use dissipates 20–26W due to conversion inefficiency and is not shown.

<table>
<thead>
<tr>
<th>Subsystem</th>
<th># resistors</th>
<th>min–max</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU core</td>
<td>3</td>
<td>0.5–27W</td>
</tr>
<tr>
<td>CPU uncore (L3, mem. controller)</td>
<td>1</td>
<td>1–9W</td>
</tr>
<tr>
<td>integrated graphics</td>
<td>2</td>
<td>n.a.</td>
</tr>
<tr>
<td>discrete graphics</td>
<td>2</td>
<td>≈15.3W</td>
</tr>
<tr>
<td>memory</td>
<td>2</td>
<td>1–5W</td>
</tr>
<tr>
<td>CPU fan</td>
<td>1</td>
<td>≈0.7W</td>
</tr>
<tr>
<td>SATA</td>
<td>3</td>
<td>1.3–3.6W</td>
</tr>
<tr>
<td>LAN</td>
<td>1</td>
<td>≈0.95W</td>
</tr>
<tr>
<td>Chipset + other</td>
<td>0</td>
<td>0.5–5W</td>
</tr>
<tr>
<td>12V ATX in</td>
<td>1</td>
<td>23–67W</td>
</tr>
</tbody>
</table>

Instead, we first identified the platform subsystems that were of interest to us and simply instrumented the resistors for those subsystems and connected them to two 32-input National Instrument USB6218 DAQs. Finally to measure the total power consumption at the wall we use a commercial WattsUp meter. The Calpella platform is powered solely by a 12-V input from the ATX power connector and we consider the 12-V rail to represent the total system power in order to eliminate the influence of variable power-supply inefficiencies. The power breakdown of the various subsystems is shown in Table 5.1.

The 16-bit NI-DAQs have a worst case error of 0.02%, but to scale the measured voltages to the range of the NI-DAQ we use precision voltage dividers, which in turn introduce a 0.035% measurement error, leading to an overall error of 0.04%. To minimize measurement overhead, the data from the NI-DAQs and the WattsUp meter is collected on a separate computer.

Along with the (externally) collected power readings, we collect OS-level statistics, hardware states and performance counters from the platform itself (for simplicity we will use the terms counter and state interchangeably). We extract OS-level statistics from /proc and /sys in Linux; these include processor utilization, disk usage, and processor C-state residency statistics. We collect hardware performance counters using the Linux perf_event framework.

By default the perf_event framework enables access to the four programmable
Figure 5.1: Power breakdown for sample workloads.

and the six fixed hardware counters available per core. As a departure from previous processor models, Nehalem processors introduce “uncore” counters, which measure the performance of the L3 cache, QPI bus, and the memory controller. To replicate the MANTIS model, we need to measure last-level cache misses in the L3 cache. Fortunately, we have a kernel patch that provides access to the eight programmable per-socket performance counters. The measurement framework reports a total of 884 counters. While it would be ideal to measure them all concurrently and allow the models to pick out the most relevant features, the small number of programmable counters that can be read concurrently makes this task impossible.

Instead, we use a simple heuristic to reduce this number to a more manageable size: we sweep through the entire set of possible counters, making sure to get at least one run for each counter; then we compute the correlation of each counter with the total system power and discard all the counters that show no variation (with power), or that have very poor correlation. This brings down the set of potential counters to about 200, which is still large. To bridge the gap, we select all of the OS counters (these can be measured concurrently), and we greedily add as many hardware counters, in order of
their correlation coefficients, as we can measure concurrently.

Note that due to issues with the aging OS required for NI-DAQ driver support, our test harness is initiated from an external machine. A high-level diagram of the measurement setup is shown in Figure 5.2. We have set up the NI-DAQ to sample each ADC channel at 10Khz and output average power consumption for each subsystem once per second for accurate power measurements. In our setup, we thus collect power readings as well as the on-platform measurements at a one-second granularity. Adjusting the collection granularity does not appreciably impact the prediction accuracy, and we feel that one second is a reasonable compromise: sampling the OS level counters at a faster rate would incur a higher overhead and introduce stronger measurement artifacts (where the act of measuring itself takes a non-trivial amount of power), while sampling it any slower might limit how quickly applications can react to changes in power consumption.

5.5.1 Benchmarks

To systematically exercise all possible states of the platform, particularly the subsystems that we are measuring, we selected an array of benchmarks from two well
known benchmarking suites, as well as a few additional benchmarks to extend the subsystem coverage. We include the majority of the benchmarks in the SpecCPU benchmark suite [35]. We include 22 of the 32 benchmarks, excluding ten because they would either not compile with a modern compiler or tended to exhaust memory. While the SpecCPU suite is a well established benchmark, it only contains single-threaded benchmarks. To get a more representative set of benchmarks that would better exercise our multi-core system, we included the PARSEC [17] benchmark suite; this consists of a range of multi threaded “emerging workloads”, including file de-duplication and x264 compression. We also include the Bonnie I/O benchmark, a parallel LinuxBuild kernel compile, StressAppTest [116], a memcached workload, as well as a synthetic cpuload benchmark. The StressAppTest program is a burn-in program designed to place a realistic high load on a system to test the hardware devices. We observe that while most of these benchmarks use the hardware as quickly as possible, evidence suggests that systems are not always fully loaded [14]. To capture this behavior, we supply memcached with a variety of different request rates to target different utilizations, and we duty cycle our synthetic floating point cpuload benchmark. Finally, we include Sleep to represent the system at idle.

5.5.2 Modeling evaluation

The effectiveness of a model is often decided by learning the model from a training set of data, and then assessing its predictive performance on a (different) testing set. Selecting the training set is often a non-trivial task and must ensure that the training set includes enough samples from various operating points. When this is not done judiciously the testing error can be large, even though the training error is small. The ideal scenario is to identify a set of “basis” benchmarks that are known to provide the sufficient coverage and to generate the training data from these benchmarks (a form of this was done in [113]). However, this is hard to achieve when systems are complex and have a large operating space. When we tried such an approach, the results were disappointing and led us to ask a more basic question: how well does the model work when the testing and training data are similar? This puts the focus on whether good models can be generated at all, rather than picking the smallest set of workloads needed
to construct a good model. We employ a well known validation technique known as $k \times 2$ cross-validation. For this technique, we randomize the ordering of the data (collected from all the benchmarks), partition into two halves, use the first half as training data and learn the model, and then compute the prediction error on the latter half. The process is repeated multiple times (we repeat 10 times) and the errors from each run are aggregated.

We note that in our experimental evaluation, the error observed on the testing data set is approximately the same as that on the training data set, not only when the error is low, but also when the error is high. This raises our confidence that the models obtained each time are sufficiently general. In the next section, we present results from building models on specific platform subsystems.

5.6 Results

In this section we present results from evaluating the various models on several different operating configurations. The metric we use to test the efficacy of a model is mean relative error, which we shorten to error in the discussion, defined as follows:

$$error = \frac{1}{n} \sum_{i=1}^{n} \left| \frac{\hat{p}_i - p_i}{p_i} \right|$$

where $\hat{p}_i$ is the model predicted power, and $p_i$ is the actual power measurement. For the results shown in this section, we execute each benchmark and configuration five times ($n = 20$). We note that the metric used is consistent with previous work [113]. One point of departure with previous work is that we measure “system power” after the PSU, and hence we capture a more accurate reflection of the actual power being consumed.

After running a large number of experiments across a variety of configurations, we can reduce the findings into three takeaways, which we discuss next.

5.6.1 Single core

To reduce number of variables we first limit the system to use one processor core only and run all the benchmarks on that single core. HyperThreading and TurboBoost
were also turned off, and the processor P-state (frequency) was fixed. This is a reasonable approximation to the systems that were used to develop the MANTIS model. For this configuration, we note the following: the mean prediction error (averaged over all the benchmarks to obtain a single number) for total system power is between 1–3%. The errors are also low for platform subsystems: for example the average error in CPU power prediction is 2–6%, and the average error in predicting memory power is about 4–8% across workloads.

**Figure 5.3:** Modeling accuracy for total system power (Single-Core). Mean relative error is 1–3% across workloads.

**Figure 5.4:** Modeling accuracy for CPU power (Single-Core). Mean relative error is 2–6% across workloads.

**Figure 5.5:** Modeling accuracy for memory power (Single-Core). Mean relative error is 4–8% across workloads.
4–8% across the different models. Due to space constraints we do not show the results for the other subsystems, however the prediction errors is similar to that of the CPU.

In this context, mean prediction error can be misleading because the mean is computed over all time samples and is weighted towards longer-running benchmarks. Thus, the error will be worse if the system is executing a specific workload that is poorly predicted.

Figure 5.3 compares the prediction error in total system power over all the benchmarks used. The models do rather well, the majority of benchmarks show an error of less than 5%. The MANTIS model does well (which we expected), but the linear-Lasso model does slightly better than do the non-linear Lasso models. Upon closer inspection we observe that this ordering of results is tied to how well the models predict the CPU power, which along with a considerable base power factor, is a large contributor to the total system power.

Figure 5.4 compares the prediction errors across different models for the CPU subsystem power. Here, the differences between the models are more pronounced. We see that the MANTIS model is off by at most a few percent for most of the benchmarks, except for canneal, StressApp, and bonnie, which have high utilization, low IPC, and consequently lower power than the model attributes to utilization alone. It is important to note that the linear-Lasso model, which picks out the set of features automatically, consistently outperforms the MANTIS model, which uses domain knowledge to select the features. Not surprisingly, the set of counters picked out by linear-Lasso is a superset of the counters used by the MANTIS model; the C-state counters included in the linear-Lasso model, but not the MANTIS model seem to improve predictive power. Thus, this goes to establish that as systems become increasingly complex, the task of applying domain knowledge to pick out the most accurate set of counters becomes progressively harder and techniques that do automatic feature selection will be very useful in building effective models.

Finally, Figure 5.5 compares the error in prediction in the memory subsystem for different models and across various benchmarks. Similar to the CPU, all models save for SVM-rbf, do quite well and have comparable errors. Also, when compared to CPU power, the prediction error for different models are similar. This hints at the fact that all
models use the same set of relevant features, which for the memory subsystem (which is simple) is quite predictable—L3-cache-misses being the most relevant and dominant feature.

Most of the results discussed so far were as expected—linear models have been shown to work well to predict full system power [41, 113]. Promisingly, even with an increase in subsystem power management complexity over prior work [18], the same linear models also do well in predicting platform subsystems.

### 5.6.2 Multicore

Next, we move to a more complex, yet more representative system configuration that utilizes all four available cores (HyperThreading and TurboBoost are still disabled and the P-state is still fixed). While more realistic, it still insulates the system from the extra hidden states of DVFS and functional contention, making power as easy to predict as possible. Figure 5.6 shows that across all the benchmarks, total system power...
is predicted to within 2-6%, which is respectable. This is well within the accuracy range required for tasks like data center server consolidation. However, as shown in Figure 5.7, the prediction error for CPU power is significantly higher, compared to that of the single core configuration, at 10-14%.

If we look more closely at the individual benchmarks for system power (Figure 5.6), we see that the error varies drastically by particular benchmark. Some benchmarks are predicted quite well (those near the left of the bar graph) and others do rather poorly (those to the right). Interestingly, the ordering of models changes with each benchmark, i.e., a particular model does not consistently do better than another over the entire set of benchmarks. In every benchmark there is at least one model that has an error less than 6%, but it is not always the same model. Our intuition for this behavior is that the model finds a roughly linear/polynomial/exponential space that fits some of the benchmarks, but then fails to capture the complex nature of contention on system resources to accurately model all workloads.

Figure 5.7 shows the prediction error across models for the CPU power. These results are even more striking. For workloads that lightly load the system (Sleep, low-rate memcached, etc.) and workloads that stress very specific components (StressAppTest, cpu1oad, etc.), the prediction is poor. This is particularly concerning because previous work shows that most production systems are run at low utilization levels [14]. Hence, one might hope that prediction is much better at idle or low load (which is a more realistic scenario in production systems). Unfortunately, the error climbs above 80% for most of the models. Thus, we find that all the models we evaluate are limited in their ability to predict the power consumption of workloads on multicore systems.

The metric of mean absolute error indicates instantaneous prediction error. An
astute reader might observe that some prediction applications might be concerned with long term averages and that the instantaneous errors might balance out. While time averaging can reduce the overall impact, many of the benchmarks experience one-sided errors and the models systematically overpredict for some and underpredict for others. Furthermore, even though the percentage error is influenced by the actual power magnitude, we find that most benchmarks are systematically mispredicted by 1–6W on average.

We posit that one of the factors that contributes in a significant way to the poor prediction performance is the increasing presence of hidden power states. In the present case, there are several resources shared across the cores (L2 caches, for one), which lead to resource contention between the cores causing bottlenecks in processing. However, this very low level behavior is not captured in any of the exported features, and consequently does not make its way into the model. Since we cannot observe the unexposed CPU state to understand what is really happening, in the following section we use the increased internal complexity of SSDs vs. hard drives, instead, to demonstrate degradation in modeling due to hidden power states.

5.6.3 Hidden states

An important concern when modeling specific hardware components is whether the model, or the inputs to the model, capture all relevant aspects of the component’s operation. This derives from the tension between the increasing complexity of the component, and hiding state to present a simple and consistent interface to the OS. If the component incorporates optimizations and circuitry that affect its power draw without varying any of the counters and states that it is exporting externally, then the model, which relies completely on the externally visible features, is likely to fail. As a case in point, there is anecdotal evidence to suggest that newer processors aggressively optimize for power by turning off functional blocks inside the CPU that are not being used, or doing clock gating at a finer granularity than what is exposed on the C-states (neither of which can be easily observed or inferred by software). Another example of this phenomenon can be seen in modern SSD drives: these include a number of performance and robustness optimizations (e.g. wear leveling, page re-writing, etc.). While these
complexities are well known [24], they are not exposed via the SATA statistics, as evidenced by low mutual information with the power values. Thus, the power consumed by a given write may have more to do with the hidden state of the device than with the write itself.

To explore this systematically, we ran the same benchmarks on the same platform but with two different hard disks. The first was a conventional 2.5" WD Caviar Black 10K-RPM HDD, and the second was a newer Intel X-25M SSD. Power measurements on the drives show similar power ranges: 0.9–3.6W for the traditional platter based HDD, and 1.2–3.6W for the newer technology SSD. Note that for both disks the features that are recorded are identical, and attempt to capture the amount of work done by the OS in writing/reading from the disk.

Figure 5.8 shows the prediction errors for both the drives. The high level take-away is that the error is consistently larger for the SSD than it is for the traditional drive. Specifically, we see that across the set of benchmarks, the model predictions are off by around 7% in the case of the conventional HDD, while they are off by approximately 15% for the SSD drive. Since the features collected and examined in each case are the same, the prediction errors are clearly caused by internal state changes in the SSD that are reflected in the power draw, but not exposed in the features being tracked. This augurs poorly for power prediction models, given that hardware complexity continues to grow by leaps and bounds.

5.7 Discussion

While our results indicate that the modeling techniques we study suffer from significant prediction error, it is natural to ask 1) whether this error is in any way fundamental, or could be overcome with more sophisticated techniques, and 2) how useful the resulting predictions might be for particular dynamic power management (DPM) applications. We discuss both issues in this section. First, we present anecdotal evidence that the inherent variability between identical hardware components is likely to introduce a basic error term to any modeling based approach that cannot be solved by adding complexity into the model. Second, we discuss how useful currently achievable levels of
accuracy can be to DPM systems.

5.7.1 Variability

Power modeling is based upon an underlying presumption that the error characteristics of the model do not change over time or over instances of the platform. That is, the model can be generated as a one-time operation by training on a specific platform instance, and the model can be used to predict the power consumption for any other instance of platform with the same specifications. While minor variations in manufacturing parts is a given, historically it has not significantly affected the operating characteristics of the platform and processors.

However, the increasing complexity of modern hardware, with staggering quantities of circuitry being stuffed into ever smaller packages exaggerates the variations significantly. These variations lead to variability in power consumption both in the active and standby modes. Furthermore, the variability is only exacerbated by aging and differences in operating environment. Recent work shows that in an 80-core general purpose chip designed by Intel, the frequency, which is directly co-related with the power consumption, of individual cores varied by as much as 25–50% for different operating voltages [40]. Research in embedded systems has shown that multiple instances of a Cortex M3 micro-controller can vary in sleep power consumption by as much as $5 \times [134]$. Recent work has also demonstrated that the performance and power consumption of flash memory chips varies widely based on age, wear and write location [24].

This level of variability raises questions about the ability of power models to generalize over identical systems because they do not actually perform identically. Errors in the power model are amplified by variations across different instances. Using our measurement platform we see significant differences between two identical Core i5-540M processors. Figure 5.9 illustrates the measured power consumption of the two processors (540M-1 and 540M-2) running our cpu\texttt{load} benchmark pinned to either Core 1 or Core 2 using Linux \texttt{cpu\_sets}. We report CPU power on a single platform so that everything—the mainboard, memory, benchmark, measurement infrastructure, and the operating environment(temperature)—but the processor is constant.

As can be observed from Figure 5.9, the CPU power consumption when exe-
Figure 5.9: Variability in power consumption measured across two CPUs of the exact same type: Intel Core i5-540M. Core 1 shows a 11% variability between the two processors, while Core 2 shows 11.2% variability. Measurements are averaged across ten runs for each case, with standard deviation marked.

Executing on Core 1 for 540M-2 is 11% higher than when using 540M-1, and is similarly 11.2% higher when executing on Core 2. Note that the power consumption is averaged across ten runs on the individual cores (Core 1 or Core 2) for the two processors (540M-1 or 540M-2). We also report the standard deviation in Figure 5.9 which is measured to be less than 3.1% in all cases. Thus, if one of the models from Section 5.5 were trained on 540M-1 and applied to workloads executing on 540M-2, then a mean prediction error of 10% could translate into a 23% error and 20% prediction error translates into a 34% prediction error. Given the undeniable trend toward more complex—and therefore inherently variable—components such as processors, this fundamental accuracy gap seems likely to continue to grow. Hence, power instrumentation may be the only choice for accurate power characterization.

5.7.2 Implications for DPM

As discussed previously, the level of accuracy and the granularity (i.e., which subsystems are characterized) required for dynamic power management is strongly tied to the particular application domain. In some cases, reducing energy might be possible with only a coarse grained and approximate power consumption estimate. In other cases,
the application is likely to need a higher degree of accuracy than modeling can currently provide.

A promising way to save energy on a computing platform is by scheduling computation more optimally. This could be done by migrating threads to different cores on the same socket or on different sockets (power gating is often done at the socket level, so using additional cores on the same socket has a very small cost). The processing cores available on a platform may be homogeneous (all derived from identical parts) or heterogeneous (from disparate parts and even architectures).

When there are a multiplicity of processing cores available, we expect the power cost to be quite different, and characterizing the power for each of these cores is critical when deciding to migrate computation. As seen in Section 5.5, the power models for subsystems like the CPU can have errors of up to 40%. When the errors dwarf the actual power variations across the cores (and this is a likely scenario when the cores are not architecturally different) it is likely that the mispredictions have an adverse effect. However, when the choice is between heterogeneous components such as between a CPU or a GPU, with significantly different power characteristics, i.e., where the variation might be larger than the model errors, it might still be acceptable to rely on modeling.

In another domain, prior work on mobile devices has shown that dynamically switching between multiple heterogeneous radios, such as WiFi and Bluetooth, can in some cases double battery lifetime [105]. Choosing between different radio alternatives like these with vastly different power characteristics seems straightforward even with very poor accuracy. However, recent work has shown that modern WiFi radios, such as those based on the 802.11n MIMO standard, have many more complex states, each with different power consumption tradeoffs [34]. Accurate component-level power characterization will therefore be essential to make optimal decisions on which radio interface or computational unit—and in which mode—to use.

Finally, we note that power-aware resource scheduling is not limited to resources within the same platform. In fact the advent of abundant cloud computing resources has accelerated research into systems, such as MAUI [37], that can use both local computation (on mobile devices) and also execute code remotely in the cloud whenever needed. Currently these systems operate under the assumption that servers in the cloud
are always-powered and, hence, their energy costs are not as important as those of battery-powered mobile devices. These systems would benefit significantly from detailed power characterization on the local mobile device as well as the servers in the cloud. Using this information, the policy decisions on when to execute code locally or remotely can be more informed and therefore more optimal. The absolute amounts of energy being considered (i.e., the execution of a single function call) in code offload scenarios, however, are fairly small, so high degrees of accuracy seem essential.

5.8 Summary

The models we consider are able to predict total system power reasonably well for both single core (1–3% mean relative error) and multi-core scenarios (2–6% mean relative error), particularly when the base power of the system is high. However, for predicting subsystem power, we show that linear regression based models often perform poorly (10–14% mean relative error, 150% worse case error for the CPU) and more complex non-linear models and SVMs do only marginally better. The poor subsystem power modeling is due to increased system and device complexity and hidden power states that are not exposed to the OS. Furthermore, our measurements show surprisingly high variability in processor power consumption, for the same configuration across multiple identical dies, highlighting the fundamental challenges with subsystem power modeling. Looking forward, while modeling techniques may suffice for some dynamic power management applications, our results motivate the need for pervasive, low-cost ways of measuring instantaneous subsystem power in commodity hardware.

\textit{scc} can help to provision a good cluster for peak load, but, if the hardware does not provide power proportionality we need accurate power estimates to actuate and evaluate the dynamic power management techniques. Our results suggest that model-based power estimation may be effective for system-level dynamic power management techniques on current hardware but future trends make complicate the models’ contribution to energy efficiency in the data center.

Chapter 5, in part, is a reprint of material as it appears in the article “Evaluating the Effectiveness of Model-Based Power Characterization” by John C. McCullough, Yu-
vraj Agarwal, Jaideep Chandrashekar, Sathyanarayan Kuppuswamy, Alex C. Snoeren, and Rajesh K. Gupta which appears in Proceedings of the USENIX Annual Technical Conference, 2011. The dissertation author was the primary investigator and author of this article.
Chapter 6

Conclusion

Large-scale computer systems are the dominant platform for computation today. These systems handle a diverse set of workloads spanning pure storage to Web processing to data-mining. Each of these workloads places different demands on the hardware. Any inefficiencies in the hardware, software, or the pairing between the two, will be amplified across the entire data center. This impact makes the efficiency of the hardware and the software paramount. We presented three techniques in this dissertation that help understand and improve efficiency in data center systems.

We first demonstrated how application deployments can be more cost effective when the hardware is tailored to the application itself. We described a novel modeling technique for representing applications and hardware at a high level. We developed a tool, scc, that combines this representation with a spectrum of workloads to identify suitable low-cost hardware configurations. We showed that this technique can achieve $2 - 4.5 \times$ cost savings versus traditional rule-of-thumb approaches.

We then identified a technique for improving the performance of the software itself. Shared scalable key-value stores are frequently employed in Web services, but their shared nature can introduce a shared performance fate when unrelated services introduce heavy load on the key-value store. We observed that the nature of scalable key-value storage allows for several optimization opportunities when a batching interval is used to group multiple requests together. The batching interval can be used to improve overall system throughput in high-load scenarios, but it introduces a trade-off between high throughput and low latency. We introduced the novel idea of treating scalable
storage as a shared medium suitable for congestion control. We addressed the challenges of adopting an appropriate networking algorithm that measures latencies and congestion windows to perceived storage performance and batching intervals. We implemented Stout as a library and deployed it in real cloud software. Stout achieved $34 \times$ lower latency under light workloads when compared to an approach optimized for throughput, and scaled to a $3 \times$ larger workload when compared to an approach optimized for latency.

Finally, we looked beyond the hardware-software balance and beyond software efficiency with storage, to understand the power consumption of the software as it executes on the hardware. Accurate estimates of system power are critical in evaluating the effectiveness of power management techniques. The trend of hardware towards multicore and energy-efficiency makes the relationship between the software increasingly complex which raises concerns regarding model accuracy. We measured system-level power consumption and system utilization counters to evaluate various modeling techniques, including several that had been used previously as well as lasso regression and support vector regression. We found that total system power estimates remain acceptably accurate with contemporary base power consumption, but subsystems grow to be more inaccurate when moving from single-core execution to multi-core.

As a result, we can improve computing efficiency in data centers. scc lowers provisioning costs by aligning applications and workloads to the provisioned hardware. Stout enables more efficient storage usage and, hence, more efficient software. Finally, the power characterization and modeling work enables better system-level dynamic power optimization.

6.1 Future Work

Data center efficiency derives from many disparate fields. There is continuing work in finding efficient ways to construct, power, and cool data centers. This work includes efforts to improve performance and energy consumption for server hardware. In this dissertation, we have focused on the choice of hardware and aspects of storage performance.

There are many opportunities to better understand the relationships between
workloads and hardware resources. One route is to enhance the models to represent some of the other nuances in hardware behavior that are important in matching existing software to hardware. We often study the elements of computation in isolation, optimizing code performance on the processor, or storage, or networking. These remain valuable, but the deceleration of single core processor performance has spurred improvements in everything else. There is continued innovation in memory throughput, cross-core communication, input-output throughput, and networking throughput. As the relative performance of the components shifts, so too do the applicable assumptions we can make about any component in isolation. These improvements in performance impact the nature of the hardware balance. Low-power processors struggle to drive a 1Gbps networking link, and today’s modern processors would require tens of hard drives or SSDs to approach saturating a 10Gbps networking link. At the same time, processors running current networking stacks struggle to drive 10Gbps without special hardware support, much in the same way that block storage subsystems fell short with the dramatic improvements in non-sequential operations when moving from hard disks to SSDs. Capturing both the high and low level nuances will help us to continue to improve the tools to understand trade offs in the diverse hardware options and we will be better equipped to find the right hardware for a particular task.

Another route is to change the software to better align with existing hardware. The shifting landscape of hardware performance will continue to highlight new and interesting software bottlenecks. There are likely many points of improvement in both application code and the operating system. Some of Stout’s optimizations incorporated the pre-execution of optimizations that the remote storage software would have done already. Given a remote bottleneck, there may be other optimizations to consider in terms of pre-computation, de-duplication, or compression. This opportunity for optimization may extend beyond current application boundaries. Many have observed that a multi-core processor makes a single system appear much like a network distributed system. By the same token, the increasing network communication speeds make a distributed system appear much like a single computer. Distributed operating systems fell out of fashion when single system performance far outpaced interconnect speeds. With the closing of this performance gap, technologies involving remote direct memory ac-
cess and network interface to storage system transfers may make for useful constructs in a new class of distributed operating systems that may be able to side-step current bottlenecks.

Finding the most efficient hardware solution for a particular piece of software will most likely lay in between matching the existing software to the right hardware and tuning the software to better fit on existing hardware. In either case, working to increase efficiency trades human time and effort for dollar costs. In current data center deployments, the scale and dollar costs encourage hardware and performance optimizations. It is unlikely that a technological advance will obviate all need for hardware and performance optimization. It is more likely that trends in energy availability will place a stronger emphasis on energy efficiency. Ultimately, we must remember to consider Amdahl’s Law on the limits of optimization before investing either time or money.
Bibliography


