Workload-aware Efficient Storage Systems

Yue Cheng

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Science and Applications

Ali R. Butt, Chair
Kirk W. Cameron
Aayush Gupta
Calvin J. Ribbens
Eli Tilevich

June 22, 2017
Blacksburg, Virginia

Keywords: Storage Systems, Cloud Computing, Data Management, Key-Value Stores,
Object Stores, Flash SSDs, Efficiency, Flexibility

Workload-aware Efficient Storage Systems

Yue Cheng

(ABSTRACT)

The growing disparity in data storage and retrieval needs of modern applications is driving the proliferation of a wide variety of storage systems (e.g., key-value stores, cloud storage services, distributed filesystems, and flash cache, etc.). While extant storage systems are designed and tuned for a specific set of applications targeting a range of workload characteristics, they lack the flexibility in adapting to the ever-changing workload behaviors. Moreover, the complexities in implementing modern storage systems and adapting ever-changing storage requirements present unique opportunities and engineering challenges.

In this dissertation, we design and develop a series of novel data management and storage systems solutions by applying a simple yet effective rule—workload awareness. We find that simple workload-aware data management strategies are effective in improving the efficiency of modern storage systems, sometimes by an order of magnitude. The first two works tackle the data management and storage space allocation issues at distributed and cloud storage level, while the third one focuses on low-level data management problems in the local storage system, which many high-level storage/data-intensive applications rely on.

In the first part of this dissertation (Chapter 3), we propose and develop MBal, a high-performance in-memory object caching framework with adaptive multi-phase load balancing, which supports not only horizontal (scale-out) but vertical (scale-up) scalability as well. In the second part of this dissertation (Chapter 4 and Chapter 5), we design and build CAST (Chapter 4), a Cloud Analytics Storage Tiering solution that cloud tenants can use to reduce monetary cost and improve performance of analytics workloads. Furthermore, we propose a hybrid cloud object storage system (Chapter 5) that could effectively engage both the cloud service providers and cloud tenants via a novel dynamic pricing mechanism. In the third part of this dissertation (Chapter 6), targeting local storage, we explore offline algorithms for flash caching in terms of both hit ratio and flash lifespan. We design and implement a multi-stage heuristic by synthesizing several techniques that manage data at the granularity of a flash erasure unit (which we call a container) to approximate the offline optimal algorithm. In the fourth part of this dissertation (Chapter 7), we are focused on how to enable fast prototyping of efficient distributed key-value stores targeting a proxy-based layered architecture. In this work, we design and build ClusterOn, a framework that significantly reduce the engineering effort required to build a full-fledged distributed key-value store.

My dissertation shows that simple workload-aware data management strategies can bring huge benefit in terms of both efficiency (i.e., performance, monetary cost, etc.) and flexibility (i.e., ease-of-use, ease-of-deployment, programmability, etc.). The principles of leveraging workload dynamicity and storage heterogeneity can be used to guide next-generation storage system software design, especially when being faced with new storage hardware technologies.

Workload-aware Efficient Storage Systems

Yue Cheng

(GENERAL AUDIENCE ABSTRACT)

Modern storage systems often manage data without considering the dynamicity of user behaviors. This design approach does not consider the unique features of underlying storage medium either. To this end, this dissertation first studies how the combinational factors of random user workload dynamicity and inherent storage hardware heterogeneity impact the data management efficiency. This dissertation then presents a series of practical and efficient techniques, algorithms, and optimizations to make the storage systems workload-aware. The experimental evaluation demonstrates the effectiveness of our workload-aware design choices and strategies.

Dedicated to my family without whom this would not have been possible.
致我最亲爱的爸爸，妈妈，静静.....

Acknowledgments

Before getting down to drafting this dissertation, I have pictured a thousand times how I would thank the many people who have helped me, who have encouraged me, and who have companied me, throughout the ups and downs of the Ph.D. journey. Now is the time. So let me express my deepest and heart-felt gratitude to the following awesome individuals!

First of all, I would like to thank my advisor Ali R. Butt for bringing me into the world of experimental computer systems, and inspiring me to join academia. I still remember the first time I met Ali during my on-campus visit as a Ph.D. applicant back in Spring'11. At that point I knew literally nothing about computer systems area. But Ali encouraged me to work at DSSL and gave me sufficient space to grow from a graduate student to an independent researcher. I have learned so much from him. I have learned how to read/write a research paper. I have learned how to deliver a good presentation. I have learned how to communicate with peers and establish a good and productive collaboration relationship. All of these soft skills have been the most precious asset of mine in building up my own academic career. He is supportive and encouraging on everything I did, both academically and otherwise. He taught me to minimize the negative effect of failure and rejections. He taught me to always be confident and look at the bright side of the whole research process. There are just too much more to list on this. I would just stop here, and express my utmost gratitude to him. Without Ali, I would have never gone through this lone journey, and would have never been able to reach where I am now.

My special thanks also go to Aayush Gupta, who has been my mentor since Summer 2013. Aayush is the most talented systems researcher I have ever met. I have been collaborating with him on almost all research projects throughout my Ph.D. He offered me a 2013 summer research internship at IBM Research Almaden. It was this internship which established the foundation of my dissertation. I will never forget the many nights he sat with me discussing our research projects at IBM Almaden. I now still keep the printed paper draft full of his hand-written comments. He always motivated me with his far-seeing vision and his persistence in solving hard research problems. He taught me the best way to sell a research idea in both writing and presentation. I owe him a huge "thank you".

I would also like to thank Fred Douglis for being a great mentor that a graduate student could

ever ask for. I greatly appreciate the full-semester internship opportunity that Fred offered me at EMC Princeton Office. Our collaboration has been joyful and fruitful. I still miss those summer days that we walked across the Smart Fair Square for coffee/burrito. He sets me a wonderful and excellent example of criticizing our own research ideas and being rigorous on each and every detail of the experimental results. Not only on the level of rigorousness towards detail, Fred's big picture vision has always amazed me. He can always point me to the right direction. His guidance has tremendously influenced me on every aspect of my research career during the second half of my Ph.D. I do hope that I can pass down whatever I have learned from him to my future students. Besides, Fred has always been supportive. He offered me many valuable advise ever since I met him. His extensive comments, feedback, and valuable suggestions were especially helpful in making my job application better.

Next, I would like to thank my other Ph.D. committee members: Professors Kirk Cameron, Cal Ribbens, and Eli Tilevich, for their insightful comments and feedback on my dissertation at various stages of my Ph.D. Special thanks go to Professor Cameron and Professor Tilevich for providing me with many great suggestions and extremely helpful advise on my job application.

This dissertation has benefited a great lot from the following colleagues during my three research internships (two at IBM Research Almaden in Summer'13 and '14, and one at Dell EMC Princeton Office in Summer'15+Fall'15): Anna Povzner, Wendy Belluomini, Jim Hafner, David Chambliss, and Ohad Rodeh at IBM; Phil Shilane, Grant Wallace, Michael Trachtman, and Darren Sawyer at Dell EMC. Working with these wonderful people has been a great and especially rewarding experience for me.

I have been fortunate to work with many other researchers and faculty members at Virginia Tech and other institutes. I am grateful to Hai Huang (IBM Research T.J. Watson) and Dongyoon Lee for their guidance on ClusterOn project. I would like to thank Changhee Jung for the extended brainstorming sessions and many discussions on GPU-assisted machine learning project. I owe a "thank you" to Gang Wang for his advise at different stages of my job search. I am indebted to Peter Desnoyers from Northeastern, Kai Li and Sanjeev Arora from Princeton, for the technical discussions on offline flash caching project during my internship at EMC. I would also like to thank Chris Gniady for his support on my job application and his jokes. My thanks also go to Kurt Luther for his many suggestions on how to start off a successful new career in academia, and sharing his experience on winning the NSF CAREER award. Danfeng Yao helped arrange the systems reading group seminar for me to practice my job talk. I thank her for her support.

My work as a Ph.D. student was made possible, and constantly enjoyable, due to the collaboration with my labmate, and one of my closest friends, Ali Anwar. We spent a lot of time together brainstorming about research problems, discussing different ideas, arguing about novelty, contributions, and impact of our work. We spent countless sleepless nights hack-

ing complex system software, writing code for ClusterOn project, running experiments on cloud/local testbed, and, needless to say, goofing off. I will never forget those good moments when we both got excited by an idea, as well as those bad moments when our paper(s) that we gave high expectation on got killed by reviewers (we have co-authored 5 papers and hopefully more are coming). Without Anwar, grad school life would never have been this much fun.

I am extremely fortunate to have worked with a wonderful group of labmates at DSSL. Safdar Iqbal helped contribute to a great deal of my early work that was part of this dissertation. More importantly, Safdar is a very close friend of mine, who is always a joy to share my mood with. He pointed me to Netflix, which has been so far the best resource for me to constantly improve my broken English. Best of luck on your new role in industry. A shout-out to my DSSL labmates at Virginia Tech: Guanying, Min, Alex, Hafeez, Henry, Krish, Hyogi, Luna, Nannan, Arnab, Bharti, Salman, and Sangeetha. Our social interactions have enriched my grad school life.

My Ph.D. would have been a "truely" lone journey without the company of an incredibly good set of friends. I was so fortunate to have Shuo Wang, Peng Mi, and Jia Liu as my roommates at different stages of my Ph.D. I cannot forget the many relaxed weekend nights that I spent in the house of the hospitable Bo Li and Yao Wang couple. The memory of the warm, good-smelling hotpot steam was always touching. I would like to thank the following people that made my life in Blacksburg enjoyable: Chang Liu, Yanqing Fu, Taige Wang, Xiaofu Ma, Kexiong Zeng, Lun Hai, Junpeng Wang, Maoyuan Sun, Nai-Ching Wang, Qianzhou Du, Fang Liu, Jianfang Guo, Hao Zhang, Kaixi Hou, Hao Wang, Jing Zhang, Zheng Song, Qingrui Liu, Long Cheng, Tong Zhang, and Xiaokui Shu. Now most of you have already started a new role in your life, while the rest of you are staying in Blacksburg finishing the degree. My best wishes for the future endeavors for all of you.

I would also like to thank the following people for making my summer time memorable when I was doing internship at Bay Area in CA. Xiaochong Zhang is one of my best friends, my basketball peer ever since college at BUPT, and the best guy to hang out, goof off with, and do whatever stupid things together. We still burst into laughter whenever we share the memory of those good old days that we spent together during Summer'13 and '14. I thank the Bay Area basketball peers at Microsoft basketball court for numerous fun every Saturday afternoon. I thank Mingyuan Xia for being my officemate during two consecutive internships at IBM Almaden.

I would like to further extend my gratitude to my old friends in San Antonio: Hang Su, Yiming Han, Qingji Zheng, and Xuebei An. The early days in the US would have been much boring and lonely without the company of you guys.

I would then like to thank the Virginia Tech Computer Science staff for their support:

Sharon, Megan, Teresa, Andrea, Matt, and Melanie; and our janitors Kathy and David, for providing us a nice and clean lab environment to stay and work in.

In closing, I am immensely grateful to my family, without whom this Ph.D. would have been lightweight and meaningless. My dad, Lianting Cheng, taught me entry-level engineering, English phonetics, Mathematics, and everything a man needs to possess. My mom, Aiqing Li, prepared me with the best environment to live, study, and play. They are my heroes. Their support and unconditional love has helped me go through many a storm. I would like to thank my grandma Ronglan Meng for her love throughout my childhood. I would like to thank my cousin Xiangping Qu for her encouragement and support ever since I was in middle school. She is inspirational role model, motivating me to come to the US studying Computer Science. She wrote me English emails helping me improve my English back in early 2000. She helped polish my personal statement even when she was overwhelmed with her research when she was doing her postdoc at Tufts back in 2008. My girlfriend, Jingjing, is my constant source of enthusiasm, creativity, and love. I thank her for always supporting me and making me a better person. I dedicate this dissertation to my family: Baba, Mama, and Jingjing.

Funding Acknowledgment My research was supported by National Science Foundation (CNS-1615411, CNS-1565314, CCF-0746832, CNS-1016408, CNS-1016793, CNS-1405697 and CNS-1422788), CS at Virginia Tech, an IBM faculty fellowship, and a NetApp faculty fellowship. Many thanks to all for making this dissertation possible.

Declaration of Collaboration In addition to my adviser Ali R. Butt, this dissertation has benefited from many collaborators, especially:

- Aayush Gupta contributed to the work included in Chapter 3, Chapter 4, Chapter 5, and Chapter 7 of the dissertation.
- Fred Douglis contributed to the offline flash caching work [74] in Chapter 6. Phil Shilane, Grant Wallace, and Peter Desnoyers contributed to the theoretical analysis of the offline flash caching heuristics of Chapter 6.
- Ali Anwar contributed to this dissertation in multiple joint papers [50, 51, 52, 53]. He made a great amount of contribution in both design and evaluation of the work in Chapter 7.
- M. Safdar Iqbal contributed to this dissertation in multiple joint papers [72, 73, 75]. He made contribution in the design and most of the evaluation part of the work in Chapter 4 and Chapter 5.
- Hai Huang, Dongyoon Lee, and Fred Douglis contributed to the work in Chapter 7. Especially, Hai contributed a significant amount to the overall idea and system design of ClusterOn.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The big data era has dramatically influenced almost every aspect of our modern lives. From social networks to enterprise business, from cloud storage to flash thumb drives, from massive-scale datacenter cluster to wearables and field sensors, the rapid growth of computing technologies has changed the way we work, do business, and entertain ourselves. One fundamental building block that enables the functioning of a myriad of services is the *data-intensive computing and storage system*. However, today's service stacks are evolving at a fast pace, and consist of a mix of complicated software and hardware. Different sub-systems, components, and tiers interact with each other. Non-holistic piece-by-piece optimizations have resulted in sub-optimal solutions, hence, inevitably dragging down the end-user experience, whereas the initial target was to provide high performance and easy of use. This becomes even more challenging as the scale of these systems increases to hundreds/thousands of machines/devices deployed at different geographic locations.

1.1 Motivation

With the growth of cloud platforms and services, distributed key-value stores and block-level storage solutions have also found their way into both public and private clouds. In fact, cloud service providers such as Amazon, IBM Cloud and Google App Engine, already support these storage services. Amazon's ElastiCache [9] is an automated in-memory key-value store deployment and management service widely used by cloud-scale web applications, e.g., Airbnb, and TicketLeap. With the improvement in network connectivity and emergence of new data sources such as Internet of Things (IoT) endpoints, mobile platforms, and wearable devices, enterprise-scale data-intensive analytics now involves terabyte- to petabyte-scale data with more data being generated from these sources constantly. Thus, storage allocation and data management would play a key role in overall performance improvement and cost reduction for this domain.

While cloud makes data management easy to deploy and scale, the dynamicity nature of modern data-intensive application workloads and the increasingly heterogeneous storage mediums (i.e., dynamically changing workload behaviors, and the vast variety of available storage services with different persistence, performance and capacity characteristics) present unique challenges for providing optimal runtime efficiency for upper-level applications. For instance, Facebook's memcache workload analysis [55] reports high access skew and time varying request patterns, implying existence of imbalance in datacenter-scale production deployments. This load imbalance is significantly amplified—by orders of magnitude—on cloud-based cache deployments, due to a number of reasons including key popularity skewness [172], multi-tenant resource sharing, and limited network bandwidth.

Storage type	**Capacity (GB/volume)**	**Throughput (MB/sec)**	**IOPS (4KB)**	**Cost ($/month)**
`ephSSD`	375	733	100,000	0.218×375
`persSSD`	100	48	3,000	0.17×100
	250	118	7,500	0.17×250
	500	234	15,000	0.17×500
`persHDD`	100	20	150	0.04×100
	250	45	375	0.04×250
	500	97	750	0.04×500
`objStore`	N/A	265	550	0.026/GB

Table 1.1: Google Cloud storage details (as of Jan. 2015).

As another example, Google Cloud Platform provides four different storage options as listed in Table 1.1. While `ephSSD` offers the highest sequential and random I/O performance, it does not provide data persistence (data stored in `ephSSD` is lost once the associated VMs are terminated). Network-attached persistent block storage services using `persHDD` or `persSSD` as storage media are relatively cheaper than `ephSSD`, but offer significantly lower performance. For instance, a 500 GB `persSSD` volume has about 2× lower throughput and 6× lower IOPS than a 375 GB `ephSSD` volume. Finally, `objStore` is a RESTful object storage service providing the cheapest storage alternative and offering comparable sequential throughput to that of a large `persSSD` volume. Other cloud service providers such as AWS EC2 [8], Microsoft Azure [1], and HP Cloud [19], provide similar storage services with different performance–cost trade-offs.

1.2 Workload-aware Storage Systems

To address the above issues, this dissertation proposes, designs, and implements a series novel techniques, algorithms, and frameworks, to make workload-oblivious data management workload-aware. This dissertation selects three different application scenarios—distributed key-value storage systems, cloud storage platforms, and local storage systems—targeting two

general modern workloads—internet-scale web workloads and big data analytics workloads. The overarching goal of this dissertation is to improve the efficiency and flexibility of modern storage applications by making the storage software layer fully-aware of both the workload characteristics and the underlying storage heterogeneity.

In the next section, we briefly describe the research problems, proposed research methodologies, and evaluation results of each work included in this dissertation.

1.2.1 Workload-aware Fast Memory Caching

Distributed key-value stores/caches have become the *sine qua non* for supporting large-scale web services in modern datacenters. Popular key-value (KV) caches such as Memcached [24] and Redis [31], have an impressive list of users including Facebook, Wikipedia, Twitter and YouTube, due to their *superb performance, high scalability, and ease of use/deployment*. It was reported that in-memory caching tier services more than 90% of database-backed queries for high performance I/Os [55, 172, 199]. The user experience is heavily dependant on the stability of the performance seen in the datacenters' memory caching tier.

In the first part of this dissertation, we propose MBal, a high-performance in-memory object caching framework with adaptive Multi-phase load Balancing, which supports not only horizontal (scale-out) but vertical (scale-up) scalability as well. MBal is able to make efficient use of available resources in the cloud through its fine-grained, partitioned, lockless design. This design also lends itself naturally to provide adaptive load balancing both within a server and across the cache cluster through an event-driven, multi-phased load balancer. While individual load balancing approaches are being leveraged in in-memory caches, MBal goes beyond the extant systems and offers a holistic solution wherein the load balancing model tracks hotspots and applies different strategies based on imbalance severity – key replication, server-local or cross-server coordinated data migration. Performance evaluation on an 8-core commodity server shows that compared to a state-of-the-art approach, MBal scales with number of cores and executes 2.3× and 12× more queries/second for GET and SET operations, respectively.

1.2.2 Workload-aware Cost-effective Tiered Cloud Storage

With the improvement in network connectivity and emergence of new data sources such as Internet of Things (IoT) endpoints, mobile platforms, and wearable devices, enterprise-scale *data-intensive* applications now involves terabyte- to petabyte-scale data with more data being generated from these sources constantly. Thus, *storage allocation and management* play a key role in overall performance improvement and cost reduction for this domain. On one hand, while cloud makes data analytics easy to deploy and scale, the vast variety of available storage services with different persistence, performance and capacity characteris-

tics, presents unique challenges, from the cloud tenants' perspective, for deploying big data analytics in the cloud. On the other hand, a highly heterogeneous cloud storage configuration exposes great opportunities for cloud service providers to increase his/her profit by leveraging a tiered pricing model.

In the second part of this dissertation, we propose CAST, a Cloud Analytics Storage Tiering solution that cloud tenants can use to reduce monetary cost and improve performance of analytics workloads. The approach takes the first step towards providing storage tiering support for data analytics in the cloud. CAST performs offline workload profiling to construct job performance prediction models on different cloud storage services, and combines these models with workload specifications and high-level tenant goals to generate a cost-effective data placement and storage provisioning plan. Furthermore, we build CAST++ to enhance CAST's optimization model by incorporating data reuse patterns and across-jobs interdependencies common in realistic analytics workloads. Tests with production workload traces from Facebook and a 400-core Google Cloud based Hadoop cluster demonstrate that CAST++ achieves 1.21× performance and reduces deployment costs by 51.4% compared to local storage configuration.

1.2.3 Workload-aware Endurance-optimized Flash Caching

Unlike traditional magnetic disk drives, flash devices such as solid state drives (SSDs) transfer data in one unit but explicitly *erase* data in a larger unit before rewriting. This erasure step is time-consuming (relative to transfer speeds) and it also has implications for the endurance of the device, as the number of erasures of a given location in flash is limited. Flash storage can be used for various purposes, including running a standalone file system and acting as a cache for a larger disk-based file system. Our work focuses on the latter. Optimizing the use of flash as a cache is a significantly more challenging problem than running a file system, because a cache has an additional degree of freedom: data can be stored in the cache or bypassed, but a file system must store all data. In addition, the cache may have different goals for the flash storage: maximize hit rate, regardless of the effect on flash endurance; limit flash wear-out, and maximize the hit rate subject to that limit; or optimize for some other utility function that takes both performance and endurance into account [146].

In the third part of this dissertation, we design and implement a multi-stage heuristic by synthesizing several techniques that manage data at the granularity of a flash erasure unit (which we call a container) to approximate the offline optimal algorithm. We find that simple techniques contribute most of the available erasure savings. Our evaluation shows that the container-optimized offline heuristic is able to provide the same optimal read hit ratio as MIN with 67% fewer flash erasures. More fundamentally, our investigation provides a useful approximate baseline for evaluating any online algorithm, highlighting the importance of comparing new policies for caching compound blocks in flash.

1.2.4 A Framework for Building Distributed Key-Value Stores

The growing disparity in data storage and retrieval needs of modern applications is driving the proliferation of a wide variety of distributed key-value (KV) stores. However, the complexities in implementing these distributed KV stores and adapting ever-changing storage requirements present unique opportunities and engineering challenges.

The fourth part of this dissertation tackles the above problems by presenting ClusterOn, a modular and compositional development platform that eases distributed KV store programming. ClusterOn is based on the insight that distributed KV stores share common distributed management functionalities (e.g., replication, consistency, and topology), and thus their development can be modularized and reused for building new stores. ClusterOn takes a single-server data store implementation (called a *datalet*), and seamlessly enable different services atop the datalets by leveraging pre-built control modules (called *controlets*). The resulting distributed stores can be easily extended for new types of services. We demonstrate how ClusterOn can enable a wide variety of KV store services with minimal engineering efforts. Furthermore, we deploy distributed KV stores developed by ClusterOn in a local testbed and a public cloud, and show that the KV stores perform comparably and sometimes better than state-of-the-art systems—which require significantly higher programming and design effort—and scale horizontally to a large number of nodes.

1.3 Research Contributions

From the above three aspects, *we demonstrate in this dissertation that we can improve the storage performance and cost efficiency across multiple layers in the data management stack by adding simple yet effective workload-awareness strategies.*

We explore how to mitigate the impact of load imbalance within the cloud datacenters' memory caching tier. We study the effectiveness of optimization heuristics on improving tenant utility while enhancing cloud profits for real world data analytics workloads. We also investigate how to exploit the resource heterogeneity in a typical cloud object storage setup for boosting real world multi-tenant workloads. Furthermore, we explore different algorithmic heuristics in minimizing the amount of writes to the flash SSD device while guaranteeing the best performance (e.g., read hit ratio) in the client-/server-side flash caching tier for enterprise primary storage workloads.

Overall, this dissertation proposes innovative systemic and algorithmic approaches to tackle the inefficiency and inflexibility of the data management strategies in modern storage system stack. In the following, we highlight the specific research contributions that this dissertation make.

Exploiting workload dynamicity in key-value stores In this work, we first evalu-

ate the impact of co-located multi-tenant cloud environment on cost of performance by conducting experiments on Amazon EC2 based cloud instance. Our observations stress the need to carefully evaluate the various resource assignment choices available to the tenants and develop simple rules-of-thumb that users can leverage for provisioning their memory caching tier. Second, based on our behavior analysis of Memcached in the cloud, we design and implement a fine-grained, partitioned, lockless in-memory caching sub-system MBal, which offers improved performance and natural support for load balancing. Third, we implement an adaptive load balancer within MBal that (1) determines the extent of load imbalance, and (2) uniquely applies local, decentralized as well as globally coordinated load balancing techniques, to (3) cost-effectively mitigate hotspots. While some of the load balancing techniques have been used before, MBal synthesizes the various techniques into a novel holistic system and automates the application of appropriate load balancing as needed. Fourth, we deploy our MBal prototype in a public cloud environment (Amazon EC2) and validate the design using comprehensive experimental evaluation on a 20-node cache cluster. Our results show that MBal agilely adapts based on workload behaviors and achieves 35% and 20% improvement in tail latency and throughput.

Exploiting analytics workload and cloud storage heterogeneity In this work, we employ a detailed experimental study and show, using both qualitative and quantitative approaches, that extant hot/cold data based storage tiering approaches cannot be simply applied to data analytics storage tiering in the cloud. We present a detailed cost-efficiency analysis of analytics workloads and workflows in a real public cloud environment. Our findings indicate the need to carefully evaluate the various storage placement and design choices, which we do, and redesign analytics storage tiering mechanisms that are specialized for the public cloud. Based on the behavior analysis of analytics applications in the cloud, we design CAST, an analytics storage tiering management framework based on simulated annealing algorithm, which searches the analytics workload tiering solution space and effectively meets customers' goals. Moreover, CAST's solver succeeds in discovering non-trivial opportunities for both performance improvement and cost savings. We extend our basic optimization solver to CAST$_{++}$ that considers data reuse patterns and job dependencies. CAST$_{++}$ supports cross-tier workflow optimization using directed acyclic graph (DAG) traversal. We evaluate our tiering solver on a 400-core cloud cluster (Google Cloud) using production workload traces from Facebook. We demonstrate that, compared to a greedy algorithm approach and a series of key storage configurations, CAST$_{++}$ improves tenant utility by 52.9% – 211.8%, while effectively meeting the workflow deadlines.

Managing data in container-optimized manner for better flash endurance In this work, we thoroughly investigate the problem space of offline compound object caching in flash. We identify and evaluate a set of techniques to make offline flash caching container-optimized. We present a multi-stage heuristic that approximates the offline optimal algorithm; our heuristic can serve as a useful approximate baseline

for analyzing any online flash caching algorithm. We experiment with our heuristic on a wide range of traces collected from production/deployed systems, to validate that it can provide a practical upper bound for both RHR and lifespan.

Fast Prototyping of Pluggable Distributed Key-Value Stores To the best of our knowledge, ClusterOn is first to completely decouple control plane and data plane development of KV stores and support a modular and compositional development environment. ClusterOn also provides pre-built controlets, a programming abstraction to built new controlets, and datalet templates. These enable the construction of new KV stores or the extension of existing ones with improved programmability, robustness, and flexibility. We demonstrate how ClusterOn can support a versatile choice of practical services and provide high performance. Based on five (two new and three existing KV store designs) datalets, we have built several flexible KV store-based services. For example, ClusterOn enables new services—e.g., with the AA topology and SC model—that are not provided by off-the-shelf Redis [30], SSDB [37], and Masstree [160]. Overall, it took us three person-days on average to implement each controlet (excluding the design phase) and less than one person-day to design, develop, and test each datalet. This underscores ClusterOn's ability to ease development of KV-store-based services.

1.4 Terminology

Note about terminology for readers of this dissertation: In this dissertation throughout, we use the term "memory cache" to refer to a software-managed *application-level* cache where the cached data resides in the main memory of a computer system; we use the term "flash cache" to refer to a software-managed *application-level* cache where the cached data resides in flash-based SSD devices; and "CPU cache" is referred to the built-in hardware cache used in CPUs. Throughout our work, we target the following major performance metrics including *throughput, tail latency, workload completion time, and read hit ratio*. For cost efficiency, we focus on the following metrics: *monetary cost per performance (or what we define tenant utility) for cloud tenants, profit for cloud providers, and SSD device lifespan for SSD storage service providers*. For flexibility metric, we focus on *ease-of-use, ease-of-deployment, and programmability*.

1.5 Dissertation Organization

The rest of the dissertation is organized as follows. In Chapter 2 we introduce the background technologies and state-of-the-art related work that lay the foundation of the research conducted in this dissertation. Chapter 3 presents a workload-aware high-performance memory caching system. Chapter 4 and Chapter 5 present a workload-aware data placement frame-

work with dynamic pricing support for next-generation cloud storage services. Chapter 6 describes a novel offline flash caching heuristic that improves the flash endurance by 3× while providing optimal performance guarantee. Chapter 7 introduces a flexible framework that helps reduce the required engineering effort when building a new distributed key-value storage system from scratch. Chapter 8 concludes and discusses the future directions.

Chapter 2

Background

In this chapter, we provide background required for various aspects of our dissertation. This dissertation is focused on applying different simple yet effective workload-aware strategies and techniques to existing storage solutions, to bridge the gap between modern data-intensive applications with the storage systems. This chapter summarizes the state-of-the-art research that is closely related to the major theme described above. We also compare them against our work by emphasizing the effectiveness, novelty, and benefits of proposed workload-aware techniques and algorithms in this dissertation.

2.1 Data Management in Key-Value Stores for Web Workloads

High Performance In-Memory Key-Value Stores Improving performance of in-memory key-value storage systems is the focus of much recent research [88, 150, 161, 197]. Specifically, systems such as Memcached on Tilera [59], Chronos [122] and MICA [153] use exclusively accessed per-core partitions to eliminate global lock contention. Similarly, MBal exploits per-core partitioning to improve performance. MBal, in addition, provides load balancing across servers as well as the ability to scale-up to fully exploit the multi-core architecture.

Storage/Memory Load Balancing Distributed hash tables (DHT) have been extensively used to lower the bandwidth consumption of routing messages while achieving storage load balancing in peer-to-peer networks. Virtual server [80] based approaches [95, 125, 180] have also been studied in this context. MBal differs from these works in that it focuses on adaptive and fast-reactive access load balancing for cloud-scale web workloads.

Proteus [148] is a dynamic server provisioning framework for memory cache cluster, which provides deterministic memory load balancing under provisioning dynamics. Similarly, Hwang

et al. [115] proposed an adaptive hash space partitioning approach that allows hash space boundary shifts between unbalanced cache nodes without further dividing the hash space. The associated framework relies on a centralized proxy to dispatch all requests from the web servers; and the centralized load balancer is actively involved in transferring data from old cache nodes to new ones. In contrast, MBal considers the memory utilization for cross-server migration for access load balancing. MBal also has the benefit of avoiding a centralized component that is inline with the migration; the centralized coordinator of MBal is used for directing only global load balancing when needed.

Access Load Balancing Replication is an effective way for achieving access load balancing. Distributed file systems such as Hadoop Distributed File System (HDFS) [186] place block replicas strategically for fault tolerance, better resource efficiency and utilization. At a finer granularity, SPORE [103] uses an adaptive key replication mechanism to redistribute the "heat" on hot objects to one or more shadow cache servers for mitigating the queuing effect. However, workloads can develop sustained and expanding unpredictable hotspots (i.e., hot shards/partitions) [111], which increase the overhead of maintaining key-level metadata on both the client and server side. In contrast, MBal is effective in handling such load imbalance as it employs a multi-phase adaptation mechanism with different cost-benefits at different levels.

Research has looked at handling load imbalance on the caching servers by caching a small set of extremely popular keys at a high-performance front-end server [87]. While Fan et al. [87] focus on achieving load balancing for an array of wimpy nodes by caching at a front-end server, Zhang et al. [207] propose hotspot redirection/replication using a centralized high-performance proxy placed in front of a cluster of heterogeneous cache servers. MBal tries to handle load imbalance within the existing caching servers without introducing another layer of indirection or other centralized bottlenecks. The centralized coordinator in MBal is sparingly used only when other phases are not sufficient to handle the hotspots. In our future work, we plan to investigate the use of distributed/hierarchical coordinators to further reduce the bottleneck of our existing coordinator if any.

Chronos [122] uses a greedy algorithm to dynamically re-assign partitions from overloaded threads to lightly-loaded ones to reduce Memcached's mean and tail latency. Similar to Chronos, MBal also adopts a partition remapping scheme as a temporary fix to load imbalance within a server. However, MBal has a wider scope in handling load imbalance, covering both a single server locally, as well as globally across the whole cache cluster.

Memcached community has implemented virtual buckets [42] as a library for supporting replication and online migration of data partitions when scaling out the cache cluster. Couchbase [79] uses this mechanism for smoothing warm-up transitioning and rebalancing the load. MBal uses a similar client-side hierarchical mapping scheme to achieve client-side key-to-server remapping. However, MBal differs from such work in that it uses cachelet migration across servers as a last resort only, and preserves the distributed approach of the original

Memcached except in the small number of cases when the global rebalancing is necessitated.

2.2 Cloud Storage and Data Management

As IT infrastructure continues to migrate to the cloud, advanced server technology can be used to enhance storage system performance and reduce the cost for cloud providers [140, 141, 142, 143, 145]. In the following, we provide a brief background of storage tiering, and categorize and compare previous work with our research.

Hot/Cold Data Classification-based Tiering Recent research [127, 151, 195] has focused on improving storage cost and utilization efficiency by placing hot/cold data in different storage tiers. Guerra et. al.[97] builds an SSD-based dynamic tiering system to minimize cost and power consumption, and existing works handle file system and block level I/Os (e.g., 4 – 32 KB) for POSIX-style workloads (e.g., server, database, file systems, etc.). However, the cost model and tiering mechanism used in prior approaches cannot be directly applied to analytics batch processing applications running in a public cloud environment, mainly due to cloud storage and analytics workload heterogeneity. In contrast, our work provides insights into design of a tiered storage management framework for cloud-based data analytics workloads.

Fine-Grained Tiering for Analytics Storage tiering has been studied in the context of data-intensive analytics batch applications. Recent analysis [98] demonstrates that adding a flash tier for serving reads is beneficial for HDFS-based HBase workloads with random I/Os. As opposed to HBase I/O characteristics, typical MapReduce-like batch jobs issues large, sequential I/Os [186] and run in multiple stages (map, shuffle, reduce). Hence, lessons learned from HBase tiering are not directly applicable to such analytics workloads. hatS [132] and open source Hadoop community [17] have taken the first steps towards integrating heterogeneous storage devices in HDFS for local clusters. However, the absence of task-level tier-aware scheduling mechanisms implies that these HDFS block granularity tiering approaches cannot avoid stragglers within a job, thus achieving limited performance gains if any. PACMan [49] solves this slow-tier straggler problem by using a memory caching policy for small jobs whose footprint can fit in the memory of the cluster. Such caching approaches are complementary to Cast as it provides a coarse-grained, static data placement solution for a complete analytics workload in different cloud storage services.

Cloud Resource Provisioning and Performance Modeling Researchers proposed approaches to reduce energy consumption of data centers by optimizing resource allocation for I/O-intensive applications [136, 137]. A performance model was proposed for applications on distributed parallel system to predict performance i.e. total execution time [139]. Frugal

Cloud File System (FCFS) [178] is a cost-effective cloud-based file storage that spans multiple cloud storage services. In contrast to POSIX file system workloads, modern analytics jobs (focus of our study) running on parallel programming frameworks like Hadoop demonstrate very different access characteristics and data dependencies; requiring a rethink of how storage tiering is done to benefit these workloads. PowerPack is a framework to support function-level power profiling of harddisk (HDDs, SSDs, etc.) [138]. Its extension [66, 67] investigated the impact of CPU speed on I/O performance. Other works such as Bazaar [119] and Conductor [196], focus on using job offline profiling and performance modeling of MapReduce applications to more cost-efficiently automate cloud resource deployment. Our work takes a thematically similar view — exploring the trade-offs of cloud services — but with a different scope that targets data analytics workloads and leverages their unique characteristics to provide storage tiering. Several systems [48, 167] are specifically designed to tackle flash storage allocation inefficiency in virtualization platforms. In contrast, we explore the inherent performance and cost trade-off of different storage services in public cloud environments.

Analytics Workflow Optimization A large body of research [84, 149, 152, 159, 203] focuses on Hadoop workflow optimizations by integrating workflow-aware scheduler into Hadoop or interfacing Hadoop with a standalone workflow scheduler. Our workflow enhancement is orthogonal and complements these works as well — CAST++ exploits cloud storage heterogeneity and performance scaling property, and uses opportunities for efficient data placement across different cloud storage services to improve workflow execution. Workflow-aware job schedulers can leverage the data placement strategy of CAST++ to further improve analytics workload performance.

2.3 Dynamic Cloud Pricing + Data Tiering

Storage Tiering Recent research [98, 132] demonstrates that adding a SSD tier for serving reads is beneficial for HDFS-based HBase and Hadoop. Existing implementations of cloud object stores provide mechanisms for tiered storage. OpenStack Swift supports storage tiering through Storage Policies [27]. Ceph, which exposes an object store API, has also added tiering support [3]. Our work focuses on providing insights into the advantages of dynamically priced tiered object storage management involving both cloud providers and tenants.

Cloud Pricing Researchers have also looked at cloud dynamic pricing [147, 156, 183]. CRAG [5] focuses on solving the cloud resource allocation problems using game theoretical schemes, while Londono et al. [157] propose a cloud resource allocation framework using colocation game strategy with static pricing. Ben-Yehuda et al. [46] propose a game-theoretic market-driven bidding scheme for memory allocation in the cloud. We adopt a simplified

game theoretic model where the cloud providers give incentives in the form of dynamic pricing and tenants adopt tiering in object stores for achieving their goals.

2.4 Data Management in Local Flash Caches

Here we provide a brief background of the offline caching algorithms, discuss the challenges of finding an offline optimal caching algorithm for container-based flash, and describe some previous analytical efforts.

Belady's MIN and its Limitations Belady's MIN algorithm [57] replaces elements whose next reference is the furthest in the future, and it is provably optimal with respect to the read hit ratio given certain assumptions [165, 166]. In particular, it applies in a single level of a caching hierarchy in which all blocks (or pages) *must* be inserted. For instance, it applies to demand-paging in a virtual memory environment.

Our environment is slightly different. We assume a DRAM cache at the highest level of the cache hierarchy and a flash device serving as an intermediate cache between DRAM and magnetic disks. A block that is read from disk into DRAM and then evicted from DRAM can be inserted into flash to make subsequent accesses faster, but it can also be removed from DRAM without inserting into flash ("read-around"). Similarly, writes need not be inserted into flash as long as persistent writes will be stored on disk (see §6.2.4).

Since this is not a demand-fetch algorithm, MIN is not necessarily the optimal strategy. Consider a simple 2-location cache with the following access sequence:

$$A,B,C,\ A,B,D,\ A,B,C,\ A,B,D\ ...$$

In a demand-fetch algorithm a missing block must be inserted into the cache, replacing another one; in this case the hit rate will be $\frac{1}{3}$, as B will always be replaced by C or D before the next access. With read-around it is not necessary for C and D to be inserted into cache, allowing hits on both A and B for a hit rate of $\frac{2}{3}$. We note, however, that such behavior may be emulated by a demand-fetch algorithm using one more cache location, which is reserved for those elements which would not be inserted into cache in the read-around algorithm. The hit rate for a read-around algorithm with N cache locations is thus bounded by the performance of MIN with N+1 locations, a negligible difference for larger values of N which we ignore in the remainder of the paper.

Even if MIN provides the optimal RHR, we argue below that it can write more blocks than another approach providing the same RHR with fewer erasures. for the remainder of this paper, We use MIN to refer to a variant of Belady's algorithm that does not insert a block into the cache if it will not be reread, while M^+ is a further enhancement that does not insert a block that will not be reread *prior to eviction*.

Temam [189] extends Belady's algorithm by exploiting spatial locality to take better advantage of processor caches. Gill [92] applies Belady's policy to multi-level cache hierarchies. His technique is useful for iterating across multiple runs of a cache policy. Karma [200] approximates the offline optimal MIN by leveraging application hints for informed replacement management. However, since Belady targets general local memory caching, it is not directly applicable to container-based flash caching due to the inherent difference between DRAM and flash.

Container-based Caching Algorithms Previous work shows that various container-based flash cache designs lead to different performance–lifespan trade-offs [144, 146, 174, 176, 188]. SDF [176], Nitro [144], and SRC [174] use a large write unit aligned to the flash erasure unit size to improve cache performance. RIPQ [188] leverages another level of indirection to track reaccessed photos within containers. Pannier [146] explicitly exploits hot/cold block and invalidation mixtures for container-based caching to further improve performance and reduce flash erasures. However, it is not known how much headroom in both performance and lifespan might exist for any state-of-the-art flash caching algorithms. To give a clear idea of how well an online flash caching algorithm performs, we need an offline optimal algorithm that incorporates performance and lifespan of the flash cache.

Analytical Approaches Considerable prior work has explored the offline optimality of caching problems in various contexts from a theoretical perspective. Albers et al. [47] and Brehob et al. [62] prove the NP-hardness of optimal replacement for non-standard caches. Chrobak et al. [76] prove the strong NP-completeness of offline caching supporting elements with varying sizes (i.e., costs). Neither explicitly studies the offline optimality of the flash caching problem with two goals that are essentially in conflict.

Other researchers have looked at related problems. Horwitz et al. [104] formulate the index register allocation problem to the shortest path problem with a general graph model and prove the optimality of the allocation algorithm. Ben-Aroya and Toledo [58] analyze a variety of offline/online wear-leveling algorithms for flash-based storage systems. Although not directly related to our problem, these works provide insights into the offline optimality of container-based flash caching.

Chapter 3

Workload-aware Fast Memory Caching

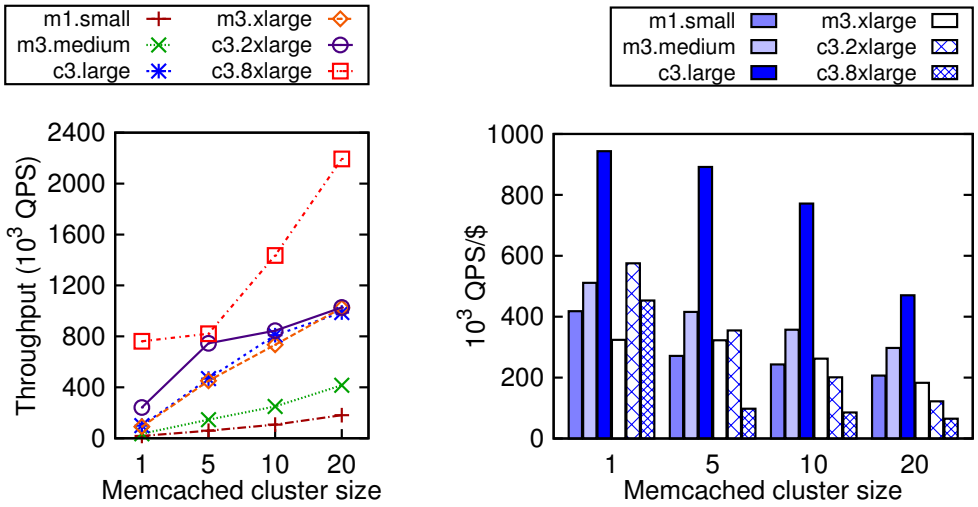
3.1 Introduction

Distributed key-value stores/caches have become the *sine qua non* for supporting today's large-scale web services. Memcached [24], a prominent in-memory key-value cache, has an impressive list of users including Facebook, Wikipedia, Twitter and YouTube. It can scale to hundreds of nodes, and in most cases, services more than 90% of database-backed queries for high performance I/Os [55, 172, 199].

With the growth of cloud platforms and services, in-memory caching solutions have also found their way into both public and private clouds. In fact, cloud service providers such as Amazon, IBM Cloud and Google App Engine, already support in-memory caching as a service. Amazon's ElastiCache [9] is an automated Memcached deployment and management service widely used by cloud-scale web applications, e.g., Airbnb, and TicketLeap.

While the cloud model makes in-memory caching solutions easy to deploy and scale, the pay-as-you-go approach leads to an important consideration for the cloud tenant: *How do I (the tenant) get the most bang-for-the-buck with in-memory caching deployment in a shared multi-tenant environment?* To understand the different aspects of this issue, we need to consider two aspects:

Impact of Resource Provisioning A key promise of the cloud model is to offer the users choice in terms of resources, services, performance, cost, tenancy, etc. In order to better understand the impact of different options in a true multi-tenant environment, we conducted an experimental study on Amazon EC2 public cloud. Figure 3.1 demonstrates the impact of scaling cluster size on Memcached performance with respect to different resource



(a) Aggregate throughput.

(b) Cost of performance.

Figure 3.1: Aggregated peak throughput and KQPS/$ observed for different Amazon EC2 cluster configurations for a 95% GET workload.

Instance type	VCPUs (cores)	Memory (GB)	Network (Gbps)	Cost ($/hr)
`m1.small`	1	1.7	0.1	0.044
`m3.medium`	1	3.75	0.5	0.07
`c3.large`	2	3.75	0.6	0.105
`m3.xlarge`	4	15	0.7	0.28
`c3.2xlarge`	8	15	1	0.42
`c3.8xlarge`	32	60	10	1.68

Table 3.1: Amazon EC2 instance details based on US West – Oregon, Oct. 2014 [7].

types (EC2 instances).

Figure 3.1(a) shows the impact on raw performance (reflected by kilo Queries Per Second (KQPS)), and Figure 3.1(b) captures the effective cost of performance by normalizing the performance with the cost of the corresponding EC instances (KQPS/$). The figures show that there is a low return on investment for powerful instances such as `c3.8xlarge` compared to relatively cheap instances such as `c3.large`. The extreme points behave as expected with the smaller-capacity instances (`m1.small`, `m3.medium`) achieving much lower throughput compared to larger-capacity instance. However, we observe that the performance of the three semi-powerful instance types (`c3.large`, `m3.xlarge`, and `c3.2xlarge`) converges to about 1.1 MQPS (million QPS) as the cluster size (for each instance type) scales to 20 nodes. We believe that this behavior can be attributed to constrained network bandwidth because of the following reasons. (1) Even though these instances have different CPU capacities (as

shown in Table 3.1), they all have similar network connectivity with an upper bound of 1 Gbps. (2) The underlying cluster or rack switches might become the bottleneck due to incast congestion [177] under Memcached's many-to-many network connection model. (3) Increasing the number of clients does not change performance. (4) The server CPUs, as observed, have a lot of free cycles in the semi-powerful instance types. For example, while CPU utilization in the `m1.small` setup was close to 100% (bounding the performance), the `c3.2xlarge` cluster setup had about 40% free cycles available. (5) Finally, improving network bandwidth to 10 Gbps (`c3.8xlarge`), doubles the throughput. This clearly shows that the performance of these semi-powerful instances is constrained by the available network bandwidth. However, even the performance of the most powerful `c3.8xlarge` instance that we tested, does not scale well with the increase in resource capacity (and monetary cost). This may be due to the multi-tenant nature of the public cloud where tenants or even virtual machines of the same tenant co-located on a host may indirectly interfere with each other's performance.

From our experiments, we infer the following. (i) While cost in the cloud scales linearly with the cluster size, the performance does not, causing the overall performance-to-cost efficiency to come down. (ii) Unlike private data centers where we typically observe large scale-out with powerful machines [55] for in-memory caching tier, cloud tenants are faced with the "problem of plentiful choices" in the form of different configurations and their impact on workloads, when deploying their in-memory caching tier. This in turn increases the number of variables tenants have to consider while making deployment decisions, and is burdensome to the extent that tenants typically choose the "easy-but-inefficient" default parameters. (iii) Our study shows that tenants may be better served by deploying moderate scale clusters with just enough CPU and memory capacity to meet their requirements to get best cost-performance ratio[1]. *This stresses the need for the caching software to make efficient use of available resources.*

Impact of Load Imbalance Facebook's memcache workload analysis [55] reports high access skew and time varying request patterns, implying existence of imbalance in datacenter-scale production deployments. This load imbalance is significantly amplified — by orders of magnitude — on cloud-based cache deployments, due to a number of reasons including key popularity skewness [172], multi-tenant resource sharing, and limited network bandwidth.

To quantify the impact of load imbalance, we measured the throughput and latency of a typical read-intensive workload (95% GET) with varying load skewness (represented by Zipfian constant). Figure 3.2 shows that the performance declines as the workload skewness increases (`unif` represents uniform load distribution). We observe that hotspots due to skewness can cause as much as 3× increase in the 99^{th} percentile tail latency and more than 60% degradation in average per-client throughput. Similar results have been observed by

[1]Finding the best combination of instance types for cloud workloads is beyond the scope of this paper and is part of our future work.

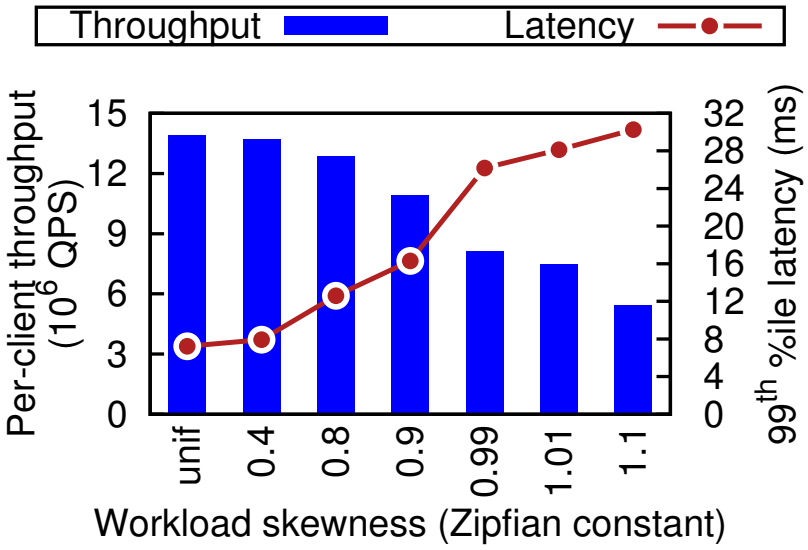


Figure 3.2: Impact of skewness on the performance of a Memcached cluster with 20 instances with 95% GET generated using Yahoo! Cloud Serving Benchmark [77] (YCSB) with 12 clients.

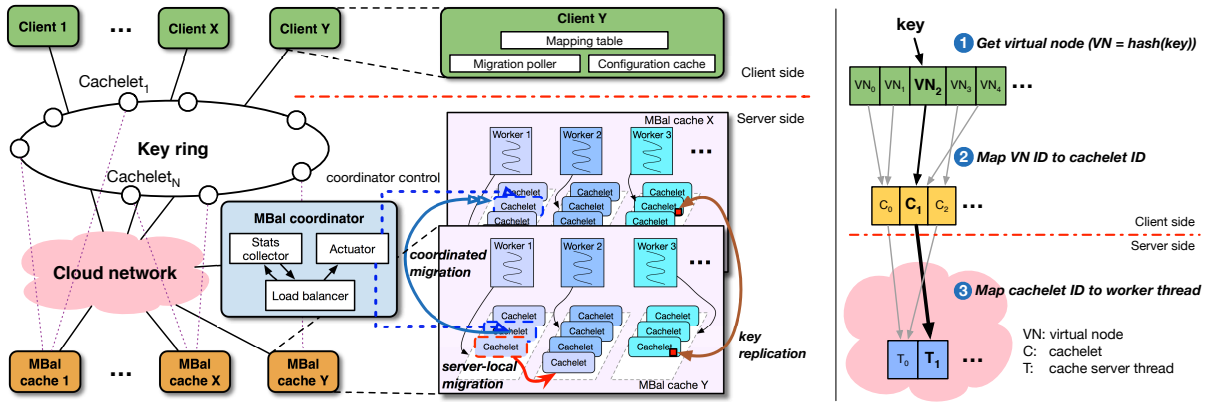
Hong et al. [103]. Thus, efficient load balancing mechanism in the caching tier is necessary for providing high performance and efficient resource (CPU, memory) utilization.

Contributions Based on the two requirements of efficient use of resources and handling load imbalance in cloud-based in-memory cache deployments, we develop MBal, an in-memory object caching framework that leverages fine-grained data partitioning and adaptive Multi-phase load Balancing. MBal performs fast, lockless inserts (SET) and lookups (GET) by partitioning user objects and compute/memory resources into non-overlapping subsets called *cachelets*. It quickly detects presence of hotspots in the workloads and uses an adaptive, multi-phase load balancing approach to mitigate any load imbalance. The cachelet-based design of MBal provides a natural abstraction for object migration both within a server and across servers in a cohesive manner.

Specifically, we make the following contributions:

1. We evaluate the impact of co-located multi-tenant cloud environment on cost of performance by conducting experiments on Amazon EC2 based cloud instance. Our observations stress *the need to carefully evaluate the various resource assignment choices available to the tenants and develop simple rules-of-thumb that users can leverage for provisioning their memory caching tier.*

2. Based on our behavior analysis of Memcached in the cloud, we design and implement



(a) MBal components. (b) Key-to-thread mapping.

Figure 3.3: MBal architecture.

a fine-grained, partitioned, lockless in-memory caching sub-system MBal, which *offers improved performance and natural support for load balancing.*

3. We implement an adaptive load balancer within MBal that (1) *determines* the extent of load imbalance, and (2) uniquely *applies* local, decentralized as well as globally coordinated load balancing techniques, to (3) *cost-effectively* mitigate hotspots. While some of the load balancing techniques have been used before, MBal *synthesizes the various techniques into a novel holistic system and automates the application of appropriate load balancing* as needed.

4. We deploy our MBal prototype in a public cloud environment (Amazon EC2) and validate the design using comprehensive experimental evaluation on a 20-node cache cluster. Our results show that MBal agilely adapts based on workload behaviors and achieves 35% and 20% improvement in tail latency and throughput.

3.2 MBal Architecture

We design MBal in light of the requirements of cloud-based deployments – efficient use of available resources and need for load balancing. Conventional object caches/stores, such as Memcached [24], use a monolithic storage architecture where key space sharding is performed at coarse server granularity while resources within an object server are shared across threads. This design has good scale-out characteristics, as demonstrated by Memcached deployments with hundreds of servers [39, 172], but is not necessarily resource efficient. For example, a known and crucial problem in Memcached is that it suffers from global lock contention, resulting in poor performance on a single server.

To address this issue, MBal performs fine-grained thread-level resource partitioning, allowing each thread within a cache server to run as a fully-functional caching unit while leveraging the benefits of running within a single address space. While the concept of thread-level resource partitioning has been explored [116, 122, 153], the approach provides significant benefits for a fast DRAM-based cache. This allows MBal to not only *scale-out* to a large number of cache nodes similar to its contemporary counterparts but also *scale-up* its performance by fully exploiting the parallelism offered by multi-core architectures. Furthermore, thread-level resource partitioning provides the ability to perform low overhead load balancing.

3.2.1 Cachelet Design

Typical in-memory object caches use consistent hashing [124] to map keys (object handles) to cache servers. The sharding process involves mapping subsets of key space to *virtual nodes* (VN) and mapping VNs to cache servers. This allows distributing non-consecutive key hashes to a server. However, the cache servers are typically unaware of the VNs.

We introduce a new abstraction, *cachelets*, to enable server worker threads to manage key space at finer granularity than a monolithic data structure. A cachelet is a configurable resource container that encapsulates multiple VNs and is managed as a separate entity by a single worker thread. As depicted in Figure 3.3(a), each worker thread in a cache server owns one or more cachelets. While the design permits one-to-one mapping between VNs and cachelets, typically there can be an order(s) of magnitude more VNs than cachelets. The choice is based on the client administrator's desired number of subsets of key space and the speed at which the load balancing algorithm should converge. To this end, cachelets help in decoupling metadata management at the servers/clients and provide resource isolation.

3.2.2 Lockless Operations

Each cachelet is bound to a single server worker thread that allocates memory, manages accesses, and maintains metadata structures and statistics for the cachelet. This partitioning ensures that in MBal, there is no lock contention or synchronization overheads across worker threads during inserts or lookups. Furthermore, this allows MBal to reduce false sharing by cross-thread resource isolation. The design is also amenable to and provides a mechanisms to quickly serialize and migrate data for load balancing (server-local and coordinated migration in Figure 3.3(a)). In the future, we aim to add functionality to cachelets such as service differentiation and server-side code execution [45], which will enable MBal to support richer services beyond object caching.

3.2.3 Key-to-Thread Mapping

A naive approach for routing a request for an object to an appropriate worker thread on a server is to use a server-side dispatcher thread: A dedicated thread on each MBal server receives client requests and dispatches the request to an appropriate worker thread based on cachelet ID in the request. We first implemented our design using this approach and quickly found the dispatcher thread to be a bottleneck. Increasing the number of dispatcher threads reduces the number of cores available on a server to service requests but does not improve performance, and thus is impractical.

To avoid this, MBal provides client-side routing capability within MBal's client library, similar to approaches used in mcrouter [172]. We associate a TCP/UDP port with each cache server worker thread so that clients can directly interact with workers without any centralized component. As shown in Figure 3.3(b), this approach performs "on-the-way-routing" via a two-level mapping table lookup on the client.

The mapping scheme enables convenient mapping changes when servers perform cachelet migration. In our implementation, we overload the field originally reserved for virtual bucket [41] in Memcached protocol header to hold cachelet ID. Thus, no client application changes are needed and web applications can easily work with MBal cache by simply linking against our Memcached protocol compliant client library. Finally, assigning a separate network port to each worker thread on a server is not a concern. This is because while the number of worker threads depends on user configuration, usually it is expected to be the same as the number of cores on the cache server machine. Note that having too many worker threads on a server can lead to significant network interrupt overhead due to request overwhelming [172] as well as cross-socket cache coherence traffic [162]. MBal also employs cache-conscious bucket lock placement suggested by [90] to reduce additional last-level cache (LLC) misses. Each bucket lock protecting one hash table bucket is co-located with that hash table entry that is cache-line-aligned. This guarantees that a hash table access results in at most one cache miss.

3.2.4 Memory Management

MBal employs hierarchical memory management using a global memory pool and thread-local memory pool managed using a slab allocator [60, 85]. Each worker thread requests memory from the global free pool in large chunks (configurable parameter) and adds the memory to its local free memory pool. New objects are allocated from thread-local slabs and object deletes return memory to thread's own pool for reuse. Furthermore, workers return memory to global heap to be reused by other threads if global free pool shrinks below a low threshold (`GLOB_MEM_LOW_THRESH`) and thread's local free pool exceeds a high threshold (`THR_MEM_HIGH_THRESH`). Such allocation reduces contention on the global heap during critical insert/delete paths and localizes memory accesses. Besides, the approach also provides

high throughput when objects are evicted (deleted) from the cache to create space for new ones. MBal uses LRU replacement algorithm similar to Memcached but aims to provide much better performance by reducing lock contention. Additionally, MBal adds support for Non-Uniform Memory Access (NUMA) aware memory allocator in the thread-local memory manager.

3.2.5 Discussion: Multi-threading vs. Multi-instance

An alternative to the multi-threaded approach in MBal is to use multiple single-threaded cache server instances that can potentially achieve good resource utilization. Such single-threaded instances can even be binded to CPU cores to reduce the overheads of process context switches. While intuitive, such a multi-instance approach is not the best design option in our opinion because of the following reasons. (1) An off-the-shelf multi-instance implementation (e.g., run multiple single-threaded cache instances) would require either static memory allocation or a dynamic per-request memory allocation (e.g., using `malloc()`). While the former approach can lead to resource under-utilization, the latter results in performance degradation due to overheads of dynamic memory allocation. (2) While possible, hierarchical memory management is costly to implement across address spaces. A multi-threaded approach allows memory to be easily rebalanced across worker threads sharing the single address space, whereas a multi-instance approach requires using shared memory (e.g., global heap in shared memory). Such sharing of memory across processes is undesirable, especially for writes and non-cache-line-aligned reads, as each such operation may suffer a TLB flush per process instead of just one in the multi-threaded case. (3) Multi-instance approach entails costly communication through either shared memory or inter-process communication, which is more expensive compared to MBal's inter-thread communication for server-local cachelet migration. (4) Recent works, e.g., from Twitter [40], have shown that multi-instance deployment makes cluster management more difficult. For example, in a cluster where each machine is provisioned with four instances, the multi-instance deployment quadruples management cost such as global monitoring. (5) Finally, emerging cloud operating systems [129, 164] are optimized for multi-threaded applications with some, such as OS^{v} [129], supporting a single address space per virtual machine. Consequently, we have designed MBal while considering all the benefits of multi-threaded approach, as well as future portability of the system.

3.3 Multi-Phase Load Balancing

MBal offers an adaptive load balancing approach that comprises different phases, each with its unique efficacy and cost. In the following, we describe these load balancing mechanisms and the design choices therein.

Phase	Key/Object replication (§3.3.2)	Server-local cachelet migration (§3.3.3)	Coordinated cachelet migration (§3.3.4)
Action	replicate hot keys across servers	migrate/swap cachelet(s) within a server	migrate/swap cachelet(s) across servers
Features	fine-grained (object/kv-pair) proportional sampling temporary (lease-based)	coarse-grained (cachelet/partition) integer linear programming (ILP) temporary (lease-based)	coarse-grained (cachelet/partition) integer linear programming (ILP) permanent
Benefit	**fast fix** for a few hot keys	**fast fix** for hot cachelet(s)	global load balancing
Limitations	home server bottleneck (for hot key writes) scalability for large hotspots	local optimization	resource consumption convergence time
Cost	**medium** extra metadata (key-level) extra space (duplicates)	**low** extra metadata (local cachelet-level)	**high** extra metadata (global cachelet-level) cross-server bulk data transfer

Table 3.2: Summary of load balancing phases of MBal.

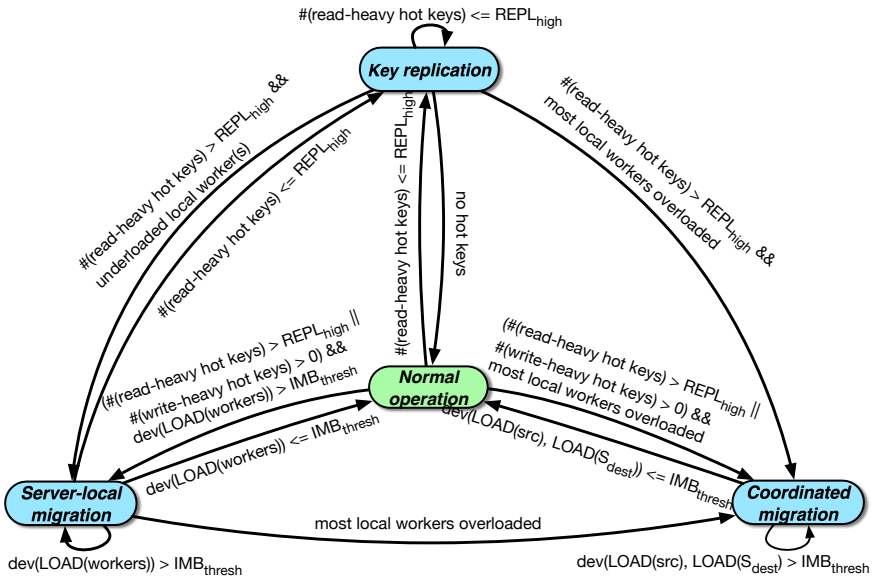


Figure 3.4: The state transitions of the MBal load balancer at each server.

3.3.1 Cluster-Wide Multi-Phase Cost/Benefit Analyzer

A distributed in-memory object cache can be load balanced using a number of techniques, such as key replication and data migration. However, each technique has a unique cost associated with it, thus requiring careful analysis to choose what technique to employ and when. The more expensive approaches typically yield better load balancing. However, such a heavy-weight approach, e.g., cross-server data migration, may not always be justified if the system is imbalanced due to a small set of hot keys. Consequently, the load balancer should consider current workload in deciding what approach to use so as to ensure high efficiency.

MBal employs an event-driven multi-phase load balancer, where each phase corresponds to a series of decision making processes triggered by event(s) based on key access patterns. A phase may also be triggered and executed simultaneously with another lower-priority phase. For example, if the configurable high replication watermark ($REPL_{high}$) is exceeded,

a worker may lower its priority on key replication by reducing the key sampling rate and thus the replication overhead. The worker may then simultaneously enter another phase, e.g., server-local cachelet migration. The goal is to generate load balancing plans that are fast and economical, yet effective for a given workload. MBal implements three phases: (1) key replication; (2) server-local cachelet migration; and (3) coordinated cachelet migration. While the first two phases are locally implemented at each MBal server providing quick solutions for ephemeral hotspots, the third phase involves a centralized coordinator to address any longer-term persistent load imbalances. Table 3.2 lists the salient properties of each of the phases, and describes the associated costs and benefits that we consider in MBal. The techniques used in each of the MBal phases are beginning to be employed in different existing systems, but individually and in an ad hoc fashion. The novelty of MBal lies in the coherent synthesis of the various load balancing techniques into a holistic design that offers an automated end-to-end system.

Figure 3.4 shows the state machine for seamlessly transitioning between the different phases of our load balancer. Each MBal server monitors the state of local workers by keeping track of both: (1) object access metrics (reads and writes) via sampling; and (2) cachelet popularity through access rates. These statistics are collected periodically (using configurable epochs) and are used to perform an online cost/benefit analysis to trigger appropriate load balancing phases. Rebalancing is triggered only if the imbalance persists across a configurable epoch dependent number, four in our implementation, of consecutive epochs. This helps to prevent unnecessary load balancing activity while allowing MBal to adapt to workload behavior shifts.

The collected information is then used to reason about the following key design questions for each load balancing phase of MBal: (1) Why is the phase necessary? (2) When is the phase triggered and what operations are undertaken? (3) What are the costs and benefits of the phase?

3.3.2 Phase 1: Key Replication

Replication of key/object offers a fine-grained mechanism to mitigate load imbalance caused by a few extremely hot keys. This is a quick fix for short ephemeral hotspots without requiring expensive cross-server data movement.

Our key replication implementation is derived from mechanisms used in SPORE [103]. Specifically, we develop our proportional sampling technique for tracking hot keys at worker granularity based on SPORE's use of access frequency and recency. Each worker has a list of other servers (and hence other workers) in the MBal cluster. A worker with a hot key (home worker) randomly selects another server as a shadow server, and replicates the hot key to one of the associated workers on the shadow server. Depending on the hotness of a key, multiple replicas of the key can be created to multiple shadow servers. Since the replicated keys do not belong to any of the cachelets of the associated shadow servers' workers,

these workers index the replicated keys using a separate (small) replica hash table. Using a separate hash table also enables shadow workers to exclude the replicated keys from being further replicated.

Upon accessing a replicated key at the home worker, a client is informed about the location of the replicas. The client can then choose any one of the replicas, which will then be used to handle *all* of the client's future read requests for that key. Writes are always performed at the home worker. Similarly as in SPORE [103], we support both synchronous and asynchronous updates. Synchronous updates have a performance overhead in the critical path, while asynchronous updates offer only eventual consistency that may result in stale reads for some clients. We leave the selection to the users based on their application needs. Furthermore, each replicated key is associated with a lease that can be renewed if the key continues to be hot or retired automatically on lease expiration. Thus, the key replication phase provides only temporary mitigation of a hotspot as the home workers for keys remain fixed.

While key replication is effective in handling short ephemeral hotspots consisting of a few keys, the approach requires both clients and MBal workers to maintain extra state of replicated keys (key locations, leases etc.). Replication also entails the using expensive DRAM for storing duplicate data. The approach also does not alleviate write-hot key based hotspots in write-intensive workloads [199]. This is because all writes are performed through a key's home worker making the home a bottleneck. To handle these limitations, multi-phased load balancer in MBal triggers other phases if the number of replicated hot keys crosses $REPL_{high}$ or if hotspots due to write-intensive workloads are observed.

3.3.3 Phase 2: Server-Local Cachelet Migration

Mitigating hotspots that consists of a large number of hot keys spanning one or more cachelets entails redistribution of the cachelets across different workers. Phase 2 represents first of the two phases of MBal, which involve such migration of cachelets. In this phase, MBal attempts to handle load imbalance at a server by triggering redistribution of cachelets to other workers running locally within the server.

As shown by the state machine in Figure 3.4, Phase 2 is triggered when there is high load imbalance between local workers (measured by absolute deviation (*dev*)) i.e., there exists idle or lightly-loaded local workers that can accommodate more load by swapping or migrating cachelets from the overloaded workers. MBal uses Algorithm 1 to bring down the load on the overloaded workers within an acceptable range, while not overwhelming the other lightly-loaded workers. The algorithm uses Integer Linear Programming (ILP) to compute an optimal migration schedule. We also ensure that the server itself is not overloaded ($SERVER\_LOAD_{thresh}$ is exceeded) before triggering Phase 2, and if so, Phase 3 is triggered instead. If ILP is not able to converge, a simple greedy algorithm will be executed eventually to reduce the *dev*.

Algorithm 1: Server-local cachelet migration.

Input: n_o: number of overloaded workers,
n_t: total number of local workers,
Set_{src}: set of source workers (local),
Set_{dest}: set of destination workers (local),
$SERVER\_LOAD_{thresh}$: server overload threshold (e.g., 75%)
Output: Cachelet migration schedule $\{m\}$

```
begin
    iter ← 0
    while True do
        if n_o/n_t > SERVER_LOAD_thresh then
            trigger Phase3
            return NULL
        if n_o == 1 then
            {m} ←SolveLP1(Set_src)
        else if n_o ≥ 2 then
            {m} ←SolveLP2(Set_src, Set_dest)
        iter ← iter + 1
        if {m} == NULL then
            iter == MAX_ITER ? return Greedy(Set_src,Set_dest) : continue
        else
            break
    return {m}
```

We define two different objective functions in our linear programming model for Phase 2. The goal of the first function is to minimize the number of migration operations with a fixed source (home) worker, while that of the second function is to minimize the load deviation across all workers.

Specifically, objective (1) is to

$$minimize \quad \sum_i \sum_j X_{ij}^k \tag{3.1}$$

$$s.t. \quad L_{*a} + \sum_i \sum_k X_{ia}^k L_i^k - \sum_i \sum_k X_{ai}^k L_a^k \leq T_a \tag{3.2}$$

$$\forall i \in S \setminus a : L_{*i} + \sum_k X_{ai}^k L_a^k - \sum_k X_{ia}^k L_i^k \leq T_i \tag{3.3}$$

where a is the index of the fixed source worker thread. Objective (2) is to

$$minimize \quad \frac{\sum_{i \in S} |L_{*i} + \sum_{j \in S \setminus i} \sum_k X_{ji}^k L_j^k - \sum_{j \in S \setminus i} \sum_k X_{ij}^k L_i^k - L_{avg}|}{N} \tag{3.4}$$

$$s.t. \quad \forall i \in S : L_{*i} + \sum_{j \in S \setminus i} \sum_k X_{ji}^k L_j^k - \sum_{j \in S \setminus i} \sum_k X_{ij}^k L_i^k \leq T_i \tag{3.5}$$

Notation	Description
X_{ij}^{k}	1 if cachelet k is migrated from worker i to worker j, 0 otherwise
T_j	maximum permissible load on worker j[2]
L_j^i	load of cachelet i on worker j
L_{*j}	total load on worker j
L_{avg}	average load of all workers
M_j^i	memory consumed by cachelet i on worker j
M_{*j}	total memory consumed by worker j
M_j	total memory capacity of worker j
S	set of workers involved in Phase 2
S_{dest}	set of workers of destination cache involved in Phase 3
N	number of workers

Table 3.3: Notations for the ILP model used in Phase 2 and Phase 3 of MBal load balancer.

Both objectives are subject to the following constraints:

$$\forall i, \forall j \in S : X_{ij} = 0 \text{ or } 1 \,, \tag{3.6}$$

$$\forall i \in S : 0 \leq \sum_{j \in S \setminus i} X_{ij} \leq 1 \,. \tag{3.7}$$

Table 3.3 describes the notations used for representing the above models. Constraints 3.2 and 3.3 are used to restrict the load after migration on the source and destination workers under a permissible limit (T_j), respectively. Similarly, constraint 3.5 restricts the load for each worker involved in optimizing objective (2). Both objectives share constraints 3.6 and 3.7, implying that migration decisions are binary and a cachelet on a particular source worker can only be migrated to one destination worker. Objective (1) is used when only a single worker is overloaded on a MBal server, and objective (2) is used for all other cases. Given the long computation time required by objective (2) to finish, we relax the objective by dividing the optimization phase into multiple iterations. In each iteration, the algorithm picks at most two overloaded threads as sources and two lightly-loaded ones as destinations. This heuristic shortens the search space for ILP, ensuring it does not affect overall cache performance and still provide good load balance (shown in §3.4.2).

MBal adopts a seamless server-local cachelet migration mechanism that incurs *near-zero* cost. Since each cache server partitions resources into cachelets that are explicitly managed by dedicated worker threads, server-local migration from one worker to another requires no data copying or transfer overheads (all workers are part of the same single address space). Only the owner worker of the cachelet is changed. Similarly as in Phase 1, we use a lease-based timeout for migrated cachelets, which means that a migrated cachelet is returned to its original home worker when the associated hotspot no longer persists. This helps address

ephemeral hotspots, while allowing for the cachelets to be restored to their home workers with negligible overhead. Clients are informed of migrated cachelets whenever the home worker receives any requests about the cachelets. Since we use leases, clients cache the home worker information and update the associated mappings to the new worker. Using the cached home worker information, clients can restore the original mapping after the lease expires.

While Phase 2 provides a lightweight mechanism to mitigate load imbalance within a server, its utility is limited by the very nature of its local optimization goals. For cases with server-wide hotspots and to provide long-term rebalancing of cachelets across MBal cluster, coordinated migration management is required.

3.3.4 Phase 3: Coordinated Cachelet Migration

An overloaded cache server or lack of an available local worker that can handle a migrated cachelet implies that Phase 2 cannot find a viable load balancing solution, thus Phase 3 is triggered. In MBal's Phase 3, cachelets from one cache server are offloaded to one or more lightly-loaded servers in the cluster.

Algorithm 2: Coordinated cachelet migration.

Input: Global cache server stats array: V_S,
input source worker: src,
IMB_{thresh}: Imbalance threshold
Output: Cachelet migration schedule V_M

```
begin
    iter ← 0
    while dev(LOAD(src), LOAD(S_dest)) > IMB_thresh && iter < MAX_ITER do
        S_dest ← min(V_S)                                   // get minimum loaded server
        V_temp ← SolveLP(src, S_dest)
        if V_temp == NULL then
            V_temp ← Greedy(src, S_dest)
        V_M ← V_M ∪ V_temp
        iter ← iter + 1
    if all cluster is hot or src still too hot then
        return NULL                                         // add new cache servers to scale out
    return V_M
```

Under Phase 3, an overloaded worker (src) notifies the centralized coordinator to trigger load balancing across cache servers. The coordinator periodically fetches statistics from all cluster workers including cachelet-level information about request arrival rate (load), amount of data stored (memory consumption), and read/write ratio.

Algorithm 2 then utilizes these statistics to choose a target server (S_{dest}) with the lowest load to redistribute cachelets from the source overloaded worker to the target's workers. The

output of the algorithm is a list of migration commands each specifying a cachelet ID and the address of a corresponding destination worker thread to which the cachelet should be migrated. The commands are then executed by the workers involved in the migration.

During each iteration, the algorithm selects the most lightly-loaded destination server and creates a candidate list of cachelets to migrate using an ILP routine whose objective is to minimize the gap between the load of source and destination workers using Equation 3.8 as follows:

$$\text{minimize} \quad \frac{\sum_{i \in S'} |L_{*i} + \sum_{j \in S' \setminus i} \sum_{k} X_{ji}^{k} L_{j}^{k} - \sum_{j \in S' \setminus i} \sum_{k} X_{ij}^{k} L_{i}^{k} - L_{avg}|}{N} \tag{3.8}$$

$$\text{s.t.}$$

$$\forall i \in S' : L_{*i} + \sum_{j \in S' \setminus i} \sum_{k} X_{ji}^{k} L_{j}^{k} - \sum_{j \in S' \setminus i} \sum_{k} X_{ij}^{k} L_{i}^{k} \leq T_i \tag{3.9}$$

$$M_{*a} + \sum_{i} \sum_{k} X_{ia}^{k} M_{i}^{k} - \sum_{i} \sum_{k} X_{ai}^{k} M_{a}^{k} \leq M_a \tag{3.10}$$

$$\forall i \in S_{dest} : M_{*i} + \sum_{k} X_{ai}^{k} M_{a}^{k} - \sum_{k} X_{ia}^{k} M_{i}^{k} \leq M_i \tag{3.11}$$

$$\forall i, \forall j \in S' : X_{ij} = 0 \text{ or } 1 \tag{3.12}$$

$$\forall i \in S' : 0 \leq \sum_{j \in S' \setminus i} X_{ij} \leq 1 \tag{3.13}$$

where a is the index of the source worker thread, and $S' = S_{dest} \cup a$.

Similar to Phase 2's ILP, constraint 3.9 bounds the load on any worker below a pre-specified permissible limit (T_i). However, unlike Phase 2, data is actually transferred across servers in Phase 3 coordinated migration. Thus, we need to ensure that the destination server has enough memory capacity to hold the migrated cachelets without causing extraneous evictions. To this end, constraints 3.10 and 3.11 ensure that the memory availability of the source and destination workers is not exceeded.

Moreover, if the ILP does not converge, – in rare cases, e.g., when the choice of a destination server is restricted in each iteration, it may not be possible to satisfy both the memory and load constraints – we employ a simple greedy solution to reduce as much load as possible from the overloaded worker.

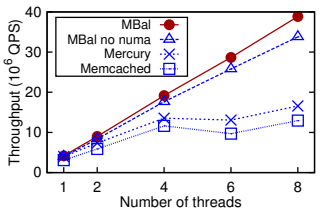
MBal maintains consistency during migration by adopting a low-overhead Write-Invalidate protocol. We employ a special migrator thread on each cache server for this purpose. Instead of migrating the entire cachelet in a single atomic operation, we migrate the tuples belonging to the selected cachelet on a per-bucket basis. We use hash table based indexes for cachelets. While keys in a bucket are being migrated, the affected worker continues serving client requests for the other buckets. Only the requests for the bucket under migration are queued. Any `UPDATE` requests to existing keys that have already been migrated result in invalidation

in both the destination (home after migration) and the source (home before migration) hash table. The `INSERT` requests for new items are treated as `NULL` operations and are directly sent to backend store. This causes no correctness problems because MBal like Memcached is a read-only, write through cache and has no dirty data. The `GET` requests are serviced for the data that is valid in the source hash table. Once the migration is complete, the source worker thread informs the coordinator about the change in the cachelet mapping. In our design, the clients exchange periodic heartbeat messages with the coordinator, and any change in cachelet mapping information is communicated to the clients by the coordinator as a response to these messages. The mapping change information only needs to be stored at the coordinator for a maximum configurable period that is slightly longer than the clients' polling period. This has two benefits: (1) it guarantees that all clients will eventually see the mapping changes within a short period of time; and (2) the coordinator is essentially stateless and only has to maintain the state information for a short time (during migration of a cachelet). Hence, after all active clients have updated their mapping tables, the coordinator informs the source worker thread to delete any metadata about its ownership of the transferred cachelets.

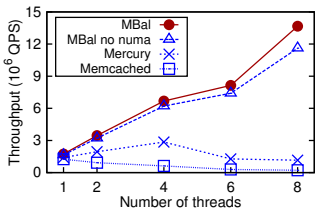
As evident by the design discussion, Phase 3 is the most expensive load balancing mechanisms employed by MBal. It serves as a last resort and is only activated when the first two phases cannot effectively mitigate the impact of load imbalance and hotspots persist. While the coordinator does not play a role during normal operation, admittedly, it can become a bottleneck for servicing the migration requests, especially if a large number of servers in the cluster are overloaded simultaneously (implying a need for adding more servers in the cluster, which is beyond the scope of the paper). Also, a failure of the coordinator during periods of imbalance can cause hotspots to persist or cache servers to maintain migration state longer (until the coordinator is brought back up). While not part of this work, we plan to exploit numerous existing projects [114, 175] in this domain to augment our coordinator design to provide more robust fault tolerance for Phase 3.

3.3.5 Discussion

While the techniques presented in this Section are applicable in any (physical or virtualized) deployment of in-memory caching clusters, use of virtualized infrastructure in the cloud is likely to demonstrate higher load imbalance and performance fluctuations due to the factors such as resource sharing across multiple tenants, compute/memory resource oversubscription, etc. Hence, MBal focuses on providing a robust in-memory caching framework for the cloud without directly dealing with multi-tenancy and resource allocation issues, which are beyond a cloud tenant's control.



(a) GET performance.



(b) SET performance.

Figure 3.5: Microbenchmark performance.

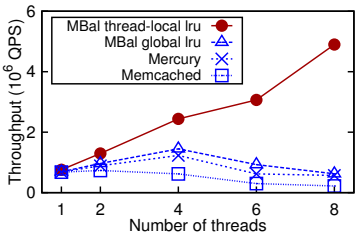
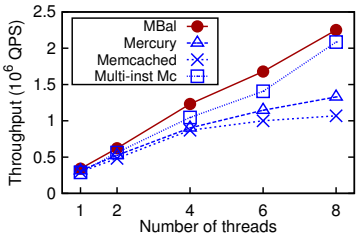
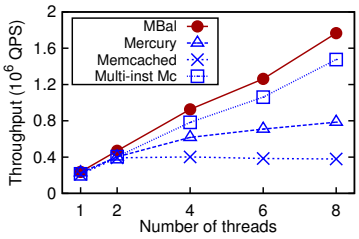


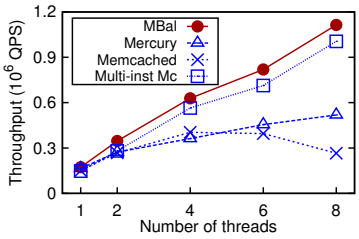
Figure 3.6: 15% cache miss.



(a) 95% GET.



(b) 75% GET.



(c) 50% GET.

Figure 3.7: Complete MBal system performance under varying GET/SET ratios.

3.4 Evaluation

MBal uses a partitioned lockless hash table as its core data structure. We support the widely used Memcached protocol (with API to support `GET`, `SET`, etc.). To this end, we extend libmemcached [23] and SpyMemcached [35] to support client-side key-to-thread mapping. The central coordinator is written in Python and uses Memcached protocol (pylibmc [29]).

We present the evaluation of MBal using a local testbed and a 20-node cluster on Amazon EC2 (`c3.large`). We evaluate the cache performance on 8-core and 32-core physical commodity servers, examine individual phases of MBal, and finally study the end-to-end system performance on a cloud cluster.

3.4.1 MBal Performance: Normal Operation

In our first set of experiments, we evaluate our design choices in building a lockless key-value cache (§3.2). We also compare the performance of MBal with existing systems such as Memcached (v1.4.19), Mercury [90] and their variants. Unless otherwise stated, we perform all tests on a dual-socket, 8-core 2.5 GHz machine with 10 Gbps Ethernet and 64 GB DRAM. Also, we enabled 2 MB hugepage support in our tests to reduce TLB misses.

Microbenchmark Performance To highlight the bottlenecks inherent in the designs of different key-value caches, we perform our next tests on a single machine. Here, each worker thread generates its own load, and there are no remote clients and thus no network traffic. First, we use two workloads, one with GET-only and the other with SET-only requests. We use a 5 GB cache, the size of which is larger than the working set of the workloads ($\approx$ 1150 MB), thus avoiding cache replacement and its effects on the results. Each workload performs a total of 32 million operations using fixed-size key-value pairs (10 B keys and 20 B values)[3]. For the GET workload, we pre-load the cache with 40 million key-value pairs, while the SET workload operates without pre-loading. The key access probability is uniformly distributed in both workloads. For fair comparison, we set the thread-local memory buffer to be the same size (256 MB) in both MBal and Mercury.

Figure 3.5(a) and Figure 3.5(b) show the throughput in terms of 10^6 QPS (MQPS) for the three systems. We observe that MBal's performance scales as the number of threads increases for both GET and SET workloads. Mercury, which uses fine-grained bucket locking, performs better than Memcached that suffers from synchronization overheads due to its coarse-grained locks. However, Mercury is not able to match the throughput of MBal due to MBal's *end-to-end* lockless design that removes most synchronization bottlenecks and allows independent resource management for each thread. Thus, for the GET workload with six threads, MBal is able to service about 2.3× more queries than Mercury, as threads in Mercury still contend for bucket-level locks.

In case of SET operations, whenever a key-value pair is overwritten, the old memory object has to be freed (pushed to a free pool). Under MBal, we garbage-collect this memory back to the thread-local free memory pool (recall that MBal transfers memory from thread-local to global free pool in bulk only under the conditions discussed in §3.2.4). In contrast, Mercury pushes the freed memory back into the global memory pool similarly as in Memcached. This introduces another synchronization overhead for write-dominant workloads in addition to the lock contention on the hash table. Thus, by mitigating both of these aspects, MBal is able to provide about 12× more throughput on the insert path for the eight workers case. In order to evaluate the impact of NUMA-based multi socket machines on cache performance, we also perform experiments on MBal with NUMA-awareness disabled. Under an 8-thread setup, MBal with NUMA-awareness achieves about 15% and 18% higher throughput for GET and SET operations, respectively, compared to MBal with no NUMA support (`MBal no numa`). The scaling trends and the widening performance gap between the studied systems as concurrency is increased (Figure 3.5) shows the benefits of MBal's lockless design and memory management policies.

Finally, we run a write-intensive workload on a 1 GB cache that is smaller than the working set, where about 15% GETs miss in the cache. Each miss triggers a SET to insert. Figure 3.6 shows that MBal with thread-local memory pools achieves about 5 MQPS. On the other

[3]Our experiments with different (larger/smaller) key and value sizes show similar trends, hence are omitted due to space constraints.

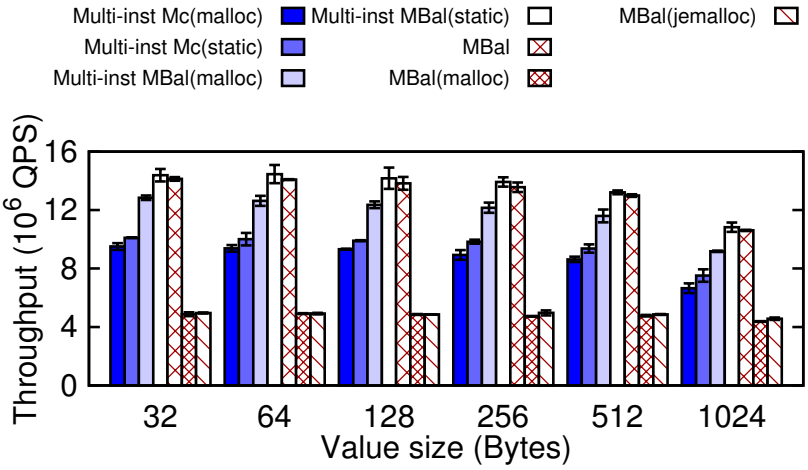


Figure 3.8: Impact of dynamic memory allocator on performance of 8 instance/thread cache. We run 100% SET workload with varying value sizes. We use `glibc`'s `malloc` and `jemalloc` as an option for cache slab allocation.

hand, MBal with only global memory pool (`MBal global lru`) achieves similar performance to Mercury and Memcached, i.e., 0.5 MQPS, which is about an order of magnitude lower than MBal with thread-local pools (`MBal thread-local lru`).

Complete Cache System Performance To test end-to-end client/server performance under MBal, we use a setup of five machines (1 server, 4 clients) with the same configuration as in §3.4.1. To amortize network overheads, we use `MultiGET` by batching 100 GETs. We use workloads generated by YCSB [77] using varying GET/SET ratios. Each workload consists of 8 million unique key-value pairs (10 B key and 20 B value) with 16 million operations generated using Zipfian key popularity distribution (Zipfian constant: 0.99). This setup helps us simulate real-world scenarios with skewed access patterns [55]. Each worker maintains 16 cachelets.

As shown in Figure 3.7, not only does MBal's performance scale with the number of worker threads for read-intensive workloads, it is also able to scale performance across workloads that are write intensive. For example, for a workload with 25% writes (Figure 3.7(b)), MBal with 8 threads outperforms both Memcached and Mercury by a factor of 4.7 and 2.3, respectively. MBal can scale up to the number of cores in the system (8 cores) for all workloads, while Memcached fails to scale with increased concurrency. This shows that as interconnects become faster, the design choices of Memcached will start to affect overall performance and MBal offers a viable alternative.

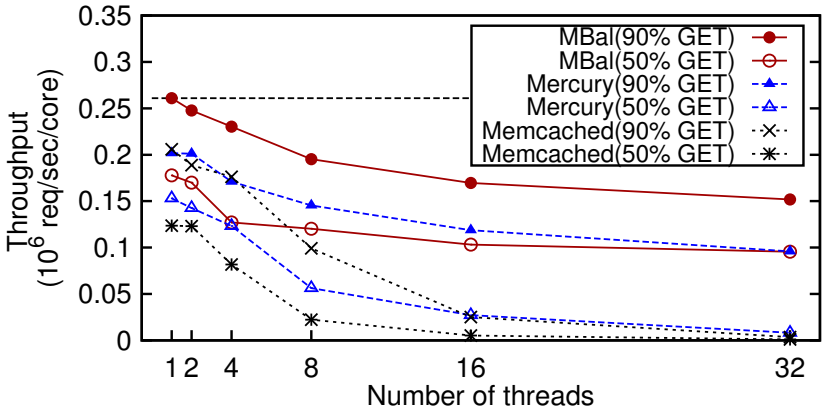


Figure 3.9: MBal scalability on a dual-socket, 32-core, 2 GHz machine. Workloads (16 B key, 32 B value) are generated from three 32-core machines in a LAN with 10 GbE interconnect, using `memaslap` with `MultiGET` enabled.

Impact of Dynamic Memory Allocation During our tests, we found that for Memcached to scale, we need to run multiple single-threaded instances (`Multi-inst Mc`). However, in our opinion, not only is such a deployment/design qualitatively inferior (§3.2.5), but as shown in Figure 3.8, it incurs significant overhead. For example, 8-instance `Multi-inst Mc(malloc)` achieves 8% less QPS on average (with value sizes ranging from 32 B to 1024 B) compared to `Multi-inst Mc(static)` due to the overhead of `malloc`. This overhead increases to 13% when we replace our optimized MBal slab allocator with `malloc`. For multi-threaded MBal, `jemalloc` does not scale due to lock contention.

Scalability on Many-core Machine Figure 3.9 demonstrates the scalability of MBal on a many-core machine. For both read-intensive (90% GET) and write-intensive (50% GET) workloads MBal achieves 18.6× and 17.2× the one-core performance, respectively, for 32 cores. The factors that limit the single-machine scalability are kernel packet processing and network interface interrupt handling. This is observed as a large fraction of CPU cycles being spent in `system mode` and servicing of `soft IRQs`[4]. As observed, both Memcached and Mercury do not scale well for write-intensive workload due to cache lock contention.

3.4.2 MBal Performance: Load Balancer

Experimental Setup We perform our next set of tests on a real Amazon EC2 cluster with 20 `c3.large` instances acting as cache servers. Clients are run on a separate cluster with up

[4]There are a number of existing works [117, 120] that we can leverage to further improve the multi-core scalability.

to 36 `c3.2xlarge` instances—that are in the same availability zone `us-west-2b`—as the cache servers. We provision both the client and server cluster instances on shared-tenant hardware. Similarly, the central coordinator of MBal's Phase 3 is run on a separate `c3.2xlarge` instance, also in the same availability zone as the servers and clients.

Performance of Individual Phases

Workloads The workloads are generated using YCSB and consists of 20 million unique key-value tuples (24 B keys and 64 B values). Each client process generates 10 million operations using the Zipfian distribution (Zipfian constant = 0.99). The workload is read-intensive with 95% GETs and 5% SETs. We run one YCSB process using 16 threads per client, and then increase the number of clients until the cache servers are fully saturated.

Phase 1: Key Replication Here, we only enable Phase 1 of MBal, and use a key sampling rate of 5%. Figure 3.10 depicts the average 99^{th} percentile read tail latency and aggregated throughput trade-off observed under our workload on different system setups. `Memcached`, `Mercury`, and `MBal (w/o load balancer)` represent the three setups of Memcached, Mercury, and MBal, respectively. `MBal (Unif)` shows the scenario with uniform workload (under MBal) and provides an upper bound for the studied metrics. We observe that without key replication support, MBal achieves only about 5% and 2% higher maximum throughput compared to the corresponding case with the same number of clients for Memcached and Mercury, respectively. This is because scaling-out the caches with associated virtualization overhead diminishes the benefits of vertical (scale-up) scalability on each individual server. However, when hot keys are replicated to one shadow server in `MBal (P1)` (key replica count = 2), the maximum throughput is observed to improve by 17%, and the 99^{th} percentile latency by 24% compared to the case with no replication. Thus, MBal is able to effectively offload the heat for hot keys, and mitigate the performance bottleneck observed in Memcached and Mercury.

Phase 2: Server-Local Cachelet Migration Next, we only turn on the Phase 2 of MBal and study its performance under our workloads. In Figure 3.10 we observe that, compared to the baseline MBal without server-local cachelet migration, Phase 2 achieves 8% higher maximum throughput and 14% lower tail latency. By migrating entire cachelets to less loaded threads, we are better able to mitigate the skew in the load. This not only helps to increase the throughput, but also improves overall latency characteristics as well. Moreover, our design ensures that the migration is effectively achieved by a simple modification of a pointer within the server, and thus incur little overhead.

Phase 3: Coordinated Cachelet Migration Figure 3.10 shows the results with only Phase 3 enabled. We observe an improvement in the maximum throughput of up to 20% and

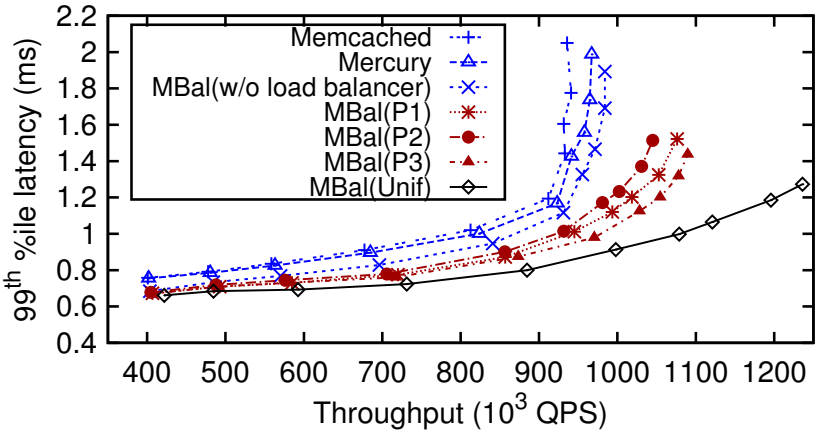


Figure 3.10: The 99^{th} percentile latency and aggregated throughput achieved by key replication (P1), server-local cachelet migration (P2), and coordinated cachelet migration (P3). We vary the number of clients from 10 to 34 to increase the throughput (shown on the X-axis). `Unif` represents uniform workload.

14%, compared to Memcached and `MBal (w/o load balancer)`, respectively. Coordinated cachelet migration also decreases the average 99^{th} percentile tail read latency by 30% and 24% compared to Memcached and `MBal (w/o load balancer)`, respectively. On the flip side, the migration incurs a high cost. We observed that migrating one cachelet at MBal cache server's peak load takes 5 to 6 seconds on average. This is the root cause of the long convergence time of Phase 3. Moreover, the CPU utilization on the central coordinator was observed to be 100% when doing the ILP computation. This shows that migration is an effective way to mitigate load imbalance. However, the increasing traffic due to migration and the increased CPU load on the centralized coordinator suggest that the approach should only be adopted for sustained hotspots and that too as a last resort. As part of our future work, we are exploring techniques for developing a hierarchical/distributed load balancer to reduce the cost of such migration.

Trade-off Analysis We have seen that for the same workload, each of the three phases of MBal are able to improve the throughput and tail latency to some extent. In our next experiment, we study the trade-offs between the different phases. Figure 3.11 plots the breakdown of read latency experienced under different cache configurations. The key replication of Phase 1 provides randomized load balancing by especially focusing on read-intensive workloads with Zipfian-like key popularity distributions. Phase 2's server-local cachelet migration serves as a lightweight intermediate stage that offers a temporary fix in response to changing workload. The limitation of Phase 2 is that it cannot offload the "heat" of overloaded cache servers to a remote server that has spare capacity. This is why the performance improve-

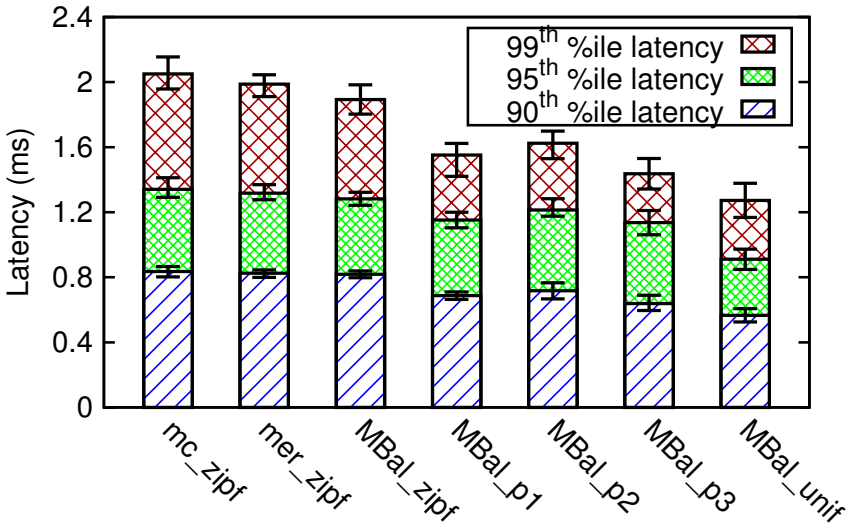


Figure 3.11: Breakdown of latency achieved under different configurations.

ment under Phase 2 is slightly less than that under Phase 1. For instance, Phase 1 is 4.2%, 5.1% and 4.4% better for 90^{th}, 95^{th} and 99^{th} percentile latency than Phase 2, respectively, as shown in Figure 3.11. Phase 3 relies on heavyweight coordinated cachelet migration—if other phases cannot achieve the desired load balance—which can optimally re-distribute load across the whole cluster and provides a better solution than the randomized key replication scheme.

Putting It All Together

In our next experiment, we evaluate the adaptivity and versatility of MBal with all three phases enabled. We use a dynamically changing workload for this test, which is generated by YCSB and is composed of three sub-workloads shown in Table 3.4. The sub-workloads are provided by the YCSB package for simulating different application scenarios. Note that, we adjust `WorkloadB` to use YCSB's hotspot key popularity distribution generator instead of the original Zipfian distribution generator. These workloads also resemble characteristics of Facebook's workloads [199].

Each sub-workload runs for 200 seconds and then switches to the next one. To get an upper-bound on performance, we also run the three workloads under uniform load distribution. We conduct two baseline runs, one under Memcached and the other under MBal with load balancer disabled. To quantitatively understand how each single phase reacts under the different workload characteristics of this test, we first perform three tests, each with one single phase enabled (similarly as for individual phase tests). Then we perform a fourth run

Workload	Characteristics	Application Scenario
`WorkloadA`	100% read, Zipfian	User account status info
`WorkloadB`	95% read, 5% update, hotspot (95% ops in 5% data)	Photo tagging
`WorkloadC`	50% read, 50% update, Zipfian	Session store recording recent actions

Table 3.4: Workload characteristics and application scenarios used for testing the multi-phase operations of MBal.

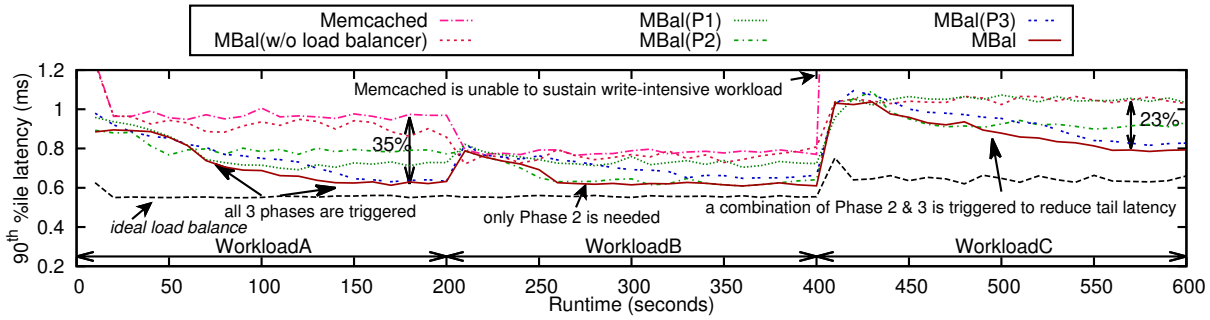


Figure 3.12: 90^{th} percentile read latency timeline for a dynamically changing workload.

with all three phases enabled to study the end-to-end behavior of MBal. Figure 3.12 plots the 90^{th} percentile read tail latency change[5]. Next, Figure 3.13 depicts the breakdown of phase triggering events that corresponds to the execution of the multi-phase test of Figure 3.12. We see that MBal reacts quickly to workload changes and employs an appropriate load balancing strategy accordingly.

Adaptivity and Versatility Under `WorkloadA`, Phase 1 and Phase 2 stabilize after around 70 s and 50 s, respectively, while it takes longer, i.e., around 150 s, for Phase 3 to stabilize the latency. Phase 3 eventually achieves slightly lower latency, compared to Phase 1, though only a limited number of cachelets are migrated. This is because, as observed before, Phase 3's optimal migration solutions perform better than randomized key replication of Phase 1. With all three phases combined, the clients see steady reduction of latency in a more smooth fashion. This is mainly due to Phase 2 serving as an effective backup approach for cache servers where key replication cannot gracefully bring the load back down to normal. As observed in Figure 3.13, Phase 3 is eventually triggered on a small number ($\approx$ 12% of all the triggered events) of servers where all worker threads are overloaded. Thus, the impact of the overhead of Phase 3 is reduced by the use of other phases. These results demonstrate the effectiveness of a multi-phase load balancing mechanism where the phases complement each other.

[5]We observed a similar trend for write latency.

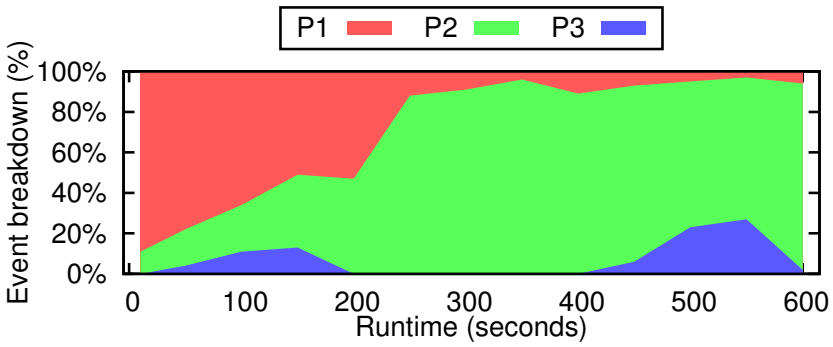


Figure 3.13: Breakdown of phase triggering events observed under the multi-phase test.

`WorkloadB` begins at 200 *s*. At this time, Phase 2's scheme immediately starts to re-balance the load. Note that Phase 1's effectiveness dramatically diminishes during this sub-workload, since hotspot distribution generator uniformly generates requests that are concentrated in 5% of all tuples. The effect of this is that the load distribution across the cache cluster is semi-uniform, whereas within each server, worker threads see non-uniformly distributed load. Phase 2 captures this behavior and promptly adapts the latency accordingly. Note that, if Phase 3 were the only available approach, it would eventually improve latency, but with a significantly long convergence duration. However, in this case, Phase 2 is triggered and ends up adapting the latency throughout the duration of `WorkloadB`. This result demonstrates that not only can Phase 2 serve as an intermediate and complementary phase for smoothing out latency variations, it can also serve as the main load balancing solution when necessary under some scenarios.

`WorkloadC` is a write-heavy sub-workload that starts at 400 *s*. Once again, Phase 1 is unable to detect hotspots as its key tracking counter uses weighted increments on read and weighted decrements on writes. This is because otherwise the overhead of propagating the writes to the key replicas would outweigh the benefits of load balancing. Here, Phase 2 can in itself effectively lower down the latency to some extent. However, Phase 3 kicks in for some of the servers to ensure that the system as a whole does not suffer from load imbalance. Thus, MBal is able to achieve its overall load-balancing goals.

Figure 3.13 shows that, unlike Phase 1 and Phase 2 that are actively invoked to balance the load throughout the three workloads, only 13% (on average) of the load balancing events involve Phase 3. This further demonstrates that Phase 3 is only sparingly used and thus is not the bottleneck in our load balancing framework.

3.5 Chapter Summary

We have presented the design of an in-memory caching tier, MBal, which adopts a fine-grained, horizontal per-core partitioning mechanism to eliminate lock contention, thus improving cache performance. It also cohesively employs different migration and replication techniques to improve performance by load balancing both within a server and across servers to re-distribute and mitigate hotspots. Evaluation for single-server case showed that MBal's cache design achieves 12× higher throughput compared to a highly-optimized Memcached design (Mercury). Testing on a cloud-based 20-node cluster demonstrates that each of the considered load balancing techniques effectively complement each other, and compared to Memcached can improve latency and throughput by 35% and 20%, respectively.

Chapter 4

Workload-aware Cost-effective Tiered Cloud Storage

From Chapter 3 we understand that managing hot/cold data at different granularities can lead to significantly improved performance and flexibility in distributed key-value storage systems. One natural question that arises in this context is, how to intelligently exploit the inherent heterogeneity that exists within both dynamically changing workloads and various storage services for interactive analytics query workloads.

In this chapter, we first conduct a comprehensive experimental analysis to study the impacts of workload heterogeneity and storage service heterogeneity on cost efficiency of analytics workloads deployed in the cloud. We then propose a new data placement framework that we call CAST, to optimize for both performance and monetary cost incurred when deploying analytics workloads in the cloud.

4.1 Introduction

The cloud computing paradigm provides powerful computation capabilities, high scalability and resource elasticity at reduced operational and administration costs. The use of cloud resources frees tenants from the traditionally cumbersome IT infrastructure planning and maintenance, and allows them to focus on application development and optimal resource deployment. These desirable features coupled with the advances in virtualization infrastructure are driving the adoption of public, private, and hybrid clouds for not only web applications, such as Netflix, Instagram and Airbnb, but also modern big data analytics using parallel programming paradigms such as Hadoop [16] and Dryad [118]. Cloud providers such as Amazon Web Services, Google Cloud, and Microsoft Azure, have started providing data analytics platform as a service [10, 15, 18], which is being adopted widely.

Storage type	Capacity (GB/volume)	Throughput (MB/sec)	IOPS (4KB)	Cost ($/month)
ephSSD	375	733	100,000	0.218×375
persSSD	100	48	3,000	0.17×100
	250	118	7,500	0.17×250
	500	234	15,000	0.17×500
persHDD	100	20	150	0.04×100
	250	45	375	0.04×250
	500	97	750	0.04×500
objStore	N/A	265	550	0.026/GB

Table 4.1: Google Cloud storage details. All the measured performance numbers match the information provided on [14] (as of Jan 2015).

With the improvement in network connectivity and emergence of new data sources such as Internet of Things (IoT) endpoints, mobile platforms, and wearable devices, enterprise-scale data-intensive analytics now involves terabyte- to petabyte-scale data with more data being generated from these sources constantly. Thus, storage allocation and management would play a key role in overall performance improvement and cost reduction for this domain.

While cloud makes data analytics easy to deploy and scale, the vast variety of available storage services with different persistence, performance and capacity characteristics, presents unique challenges for deploying big data analytics in the cloud. For example, Google Cloud Platform provides four different storage options as listed in Table 4.1. While `ephSSD` offers the highest sequential and random I/O performance, it does not provide data persistence (data stored in `ephSSD` is lost once the associated VMs are terminated). Network-attached persistent block storage services using `persHDD` or `persSSD` as storage media are relatively cheaper than `ephSSD`, but offer significantly lower performance. For instance, a 500 GB `persSSD` volume has about 2× lower throughput and 6× lower IOPS than a 375 GB `ephSSD` volume. Finally, `objStore` is a RESTful object storage service providing the cheapest storage alternative and offering comparable sequential throughput to that of a large `persSSD` volume. Other cloud service providers such as AWS EC2 [8], Microsoft Azure [1], and HP Cloud [19], provide similar storage services with different performance–cost trade-offs.

The heterogeneity in cloud storage services is further complicated by the varying types of jobs within analytics workloads, e.g., iterative applications such as `KMeans` and `Pagerank`, and queries such as `Join` and `Aggregate`. For example, in map-intensive `Grep`, the map phase accounts for the largest part of the execution time (mostly doing I/Os), whereas CPU-intensive `KMeans` spends most of the time performing computation. Furthermore, short-term (within hours) and long-term (daily, weekly or monthly) data reuse across jobs is common in production analytics workloads [49, 68]. As reported in [68], 78% of jobs in Cloudera Hadoop workloads involve data reuse. Another distinguishing feature of analytics workloads is the presence of workflows that represents interdependencies across jobs. For instance, analytics queries are usually converted to a series of batch processing jobs, where the output of one

job serves as the input of the next job(s).

The above observations lead to an important question for the cloud tenants *How do I (the tenant) get the most bang-for-the-buck with data analytics storage tiering/data placement in a cloud environment with highly heterogeneous storage resources?* To answer this question, this paper conducts a detailed quantitative analysis with a range of representative analytics jobs in the widely used Google Cloud environment. The experimental findings and observations motivate the design of CAST, which leverages different cloud storage services and heterogeneity within jobs in an analytics workload to perform cost-effective storage capacity allocation and data placement.

CAST does offline profiling of different applications (jobs) within an analytics workload and generates job performance prediction models based on different storage services. It lets tenants specify high-level objectives such as maximizing tenant utility, or minimizing deadline miss rate. CAST then uses a simulated annealing based solver that reconciles these objectives with the performance prediction models, other workload specifications and the different cloud storage service characteristics to generate a data placement and storage provisioning plan. The framework finally deploys the workload in the cloud based on the generated plan. We further enhance our basic tiering design to build CAST++, which incorporates the data reuse and workflow properties of an analytics workload.

Specifically, we make the following contributions in this paper:

1. We employ a detailed experimental study and show, using both qualitative and quantitative approaches, that extant hot/cold data based storage tiering approaches cannot be simply applied to data analytics storage tiering in the cloud.

2. We present a detailed cost-efficiency analysis of analytics workloads and workflows in a real public cloud environment. Our findings indicate the need to carefully evaluate the various storage placement and design choices, which we do, and redesign analytics storage tiering mechanisms that are specialized for the public cloud.

3. Based on the behavior analysis of analytics applications in the cloud, we design CAST, an analytics storage tiering management framework based on simulated annealing algorithm, which searches the analytics workload tiering solution space and *effectively meets* customers' goals. Moreover, CAST's solver *succeeds in discovering non-trivial opportunities* for both performance improvement and cost savings.

4. We extend our basic optimization solver to CAST++ that considers data reuse patterns and job dependencies. CAST++ supports cross-tier workflow optimization using directed acyclic graph (DAG) traversal.

5. We evaluate our tiering solver on a 400-core cloud cluster (Google Cloud) using production workload traces from Facebook. We demonstrate that, compared to a greedy

algorithm approach and a series of key storage configurations, CAST++ improves tenant utility by 52.9% – 211.8%, while effectively meeting the workflow deadlines.

4.2 A Case for Cloud Storage Tiering

In this section, we first establish the need for cloud storage tiering for data analytics workloads. To this end, we characterize the properties of applications that form a typical analytics workload and demonstrate the impact of these properties on the choice of cloud storage services. We then argue that extant tiering techniques, such as hot/cold data based segregation and fine-grained partitioning within a single job, are not adequate; *rather a course-grained, job-level storage service tiering is needed for cloud-based data analytics.*

4.2.1 Characterization of Data Analytics Workloads

We characterize the analytics workloads along two dimensions. First, we study the behavior of individual applications within a large workload when executed on parallel programming paradigms such as MapReduce — demonstrating the benefits of different storage services for various applications. Second, we consider the role of cross-job relationships (an analytics workload comprises multiple jobs each executing an application) and show how these interactions affect the choice of efficient data placement decisions for the same applications.

Experimental Study Setup

We select four representative analytics applications that are typical components of real-world analytics workloads [68, 205] and exhibit diversified I/O and computation characteristics, as listed in Table 4.2. `Sort`, `Join` and `Grep` are I/O-intensive applications. The execution time of `Sort` is dominated by the shuffle phase I/O, transferring data between mappers and reducers. In contrast, `Grep` spends most of its runtime in the map phase I/O, reading the input and finding records that match given patterns. `Join` represents an analytics query that combines rows from multiple tables and performs the join operation during the reduce phase, and thus is reduce intensive. `KMeans` is an iterative machine learning clustering application that spends most of its time in the compute phases of map and reduce iterations, which makes it CPU-intensive.

The experiments are performed in Google Cloud using a `n1-standard-16` VM (16 vCPUs, 60 GB memory) with the master node on a `n1-standard-4` VM (4 vCPUs, 15 GB memory). Intermediate data is stored on the same storage service as the original data, except for `objStore`, where we use `persSSD` for intermediate storage. Unless otherwise stated, all experiments in this section are conducted using the same compute resources but with different storage configurations as stated.

App.	I/O-intensive			CPU-intensive
	Map	Shuffle	Reduce	
Sort	✗	✓	✗	✗
Join	✗	✓	✓	✗
Grep	✓	✗	✗	✗
KMeans	✗	✗	✗	✓

Table 4.2: Characteristics of studied applications.

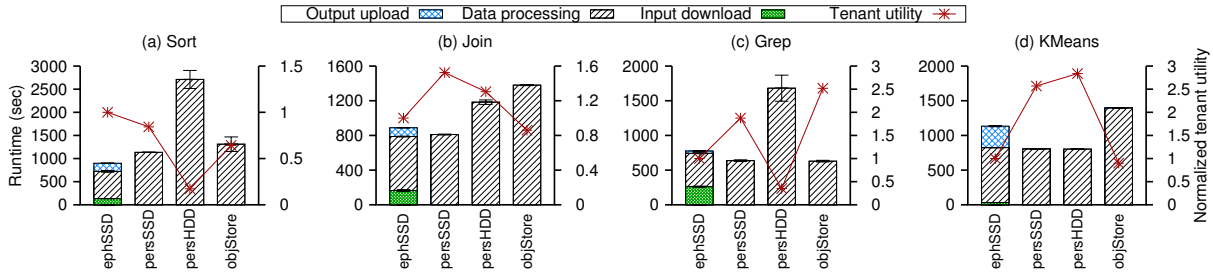


Figure 4.1: Application performance and achieved tenant utility on different cloud storage tiers.

Analysis: Application Granularity

Figure 4.1 depicts both the execution time of the studied applications and tenant utility for different choices of storage services. We define *tenant utility* (or simply "utility," used interchangeably) to be $\frac{1/execution\ time}{cost\ in\ dollars}$. This utility metric is based on the tenants' economic constraints when deploying general workloads in the cloud. Figure 4.1 (a) shows that `ephSSD` serves as the best tier for both execution time and utility for `Sort` even after accounting for the data transfer cost for both upload and download from `objStore`. This is because there is no data reduction in the map phase and the entire input size is written to intermediate files residing on `ephSSD` that has about 2× higher sequential bandwidth than `persSSD`. Thus, we get better utility from `ephSSD` than `persSSD`, albeit at a slightly higher cost. On the other hand, Figure 4.1 (b) shows that, `Join` works best with `persSSD`, while it achieves the worst utility on `objStore`. This is due to high overheads of setting up connections to request data transfers using the Google Cloud Storage Connector (GCS connector) for Hadoop APIs [13] for the many small files generated by the involved reduce tasks. `Grep`'s map-intensive feature dictates that its performance solely depends on sequential I/O throughput of the storage during the map phase. Thus, in Figure 4.1 (c) we observe that both `persSSD` and `objStore` provide similar performance (both have similar sequential bandwidth as seen in Table 4.1) but the lower cost of `objStore` results in about 34.3% higher utility than `persSSD`. Similarly, for the CPU-bound `KMeans`, while `persSSD` and `persHDD` provide similar performance, the lower cost of `persHDD` yields much better utility as shown in Figure 4.1 (d).

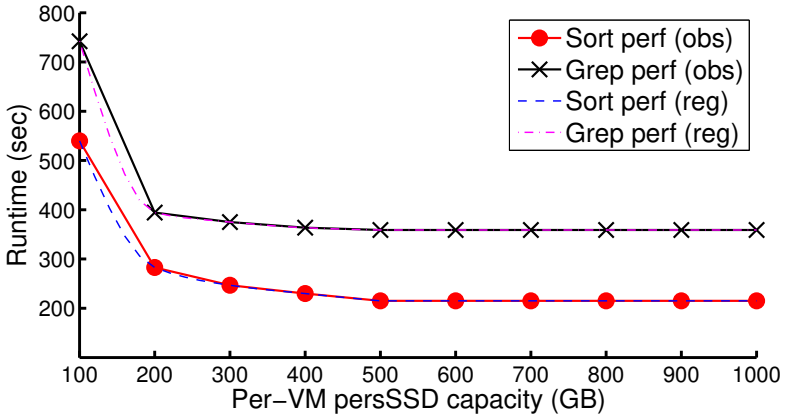


Figure 4.2: Impact of scaling `persSSD` volume capacity for `Sort` and `Grep`. The regression model is used in CAST's tiering approach and is described in detail in §4.3. These tests are conducted on a 10-VM `n1-standard-16` cluster.

Performance Scaling In Google Cloud, performance of network-attached block storage depends on the size of the volume, as shown in Table 4.1. Other clouds such as Amazon AWS offer different behavior but typically the block storage performance in these clouds can be scaled by creating logical volumes by striping (RAID-0) across multiple network-attached block volumes. In Figure 4.2, we study the impact of this capacity scaling on the execution time of two I/O-intensive applications, `Sort` and `Grep` (we also observe similar patterns for other I/O-intensive applications). For a network-attached `persSSD` volume, the dataset size of `Sort` is 100 GB and that of `Grep` is 300 GB. We observe that as the volume capacity increases from 100 GB to 200 GB, the run time of both `Sort` and `Grep` is reduced by 51.6% and 60.2%, respectively. Any further increase in capacity offers marginal benefits. This happens because in both these applications the I/O bandwidth bottleneck is alleviated when the capacity is increased to 200 GB. Beyond that, the execution time is dependent on other parts of the MapReduce framework. These observations imply that it is possible to achieve desired application performance in the cloud without resorting to unnecessarily over-provisioning of the storage and thus within acceptable cost.

Key Insights From our experiments, we infer the following. (i) There is no one storage service that provides the best raw performance as well as utility for different data analytics applications. (ii) For some applications, slower storage services, such as `persHDD`, may provide better utility and comparable performance to other costlier alternatives. (iii) Elasticity and scalability of cloud storage services should be leveraged through careful over-provisioning of capacity to reduce performance bottlenecks in I/O intensive analytics applications.

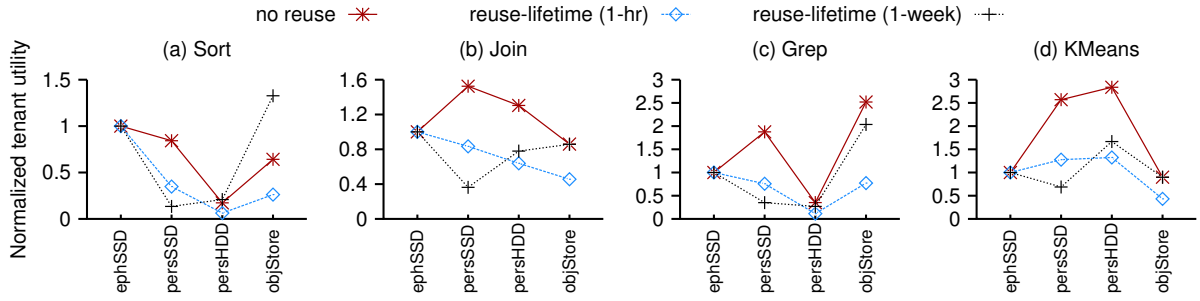


Figure 4.3: Tenant utility under different data reuse patterns.

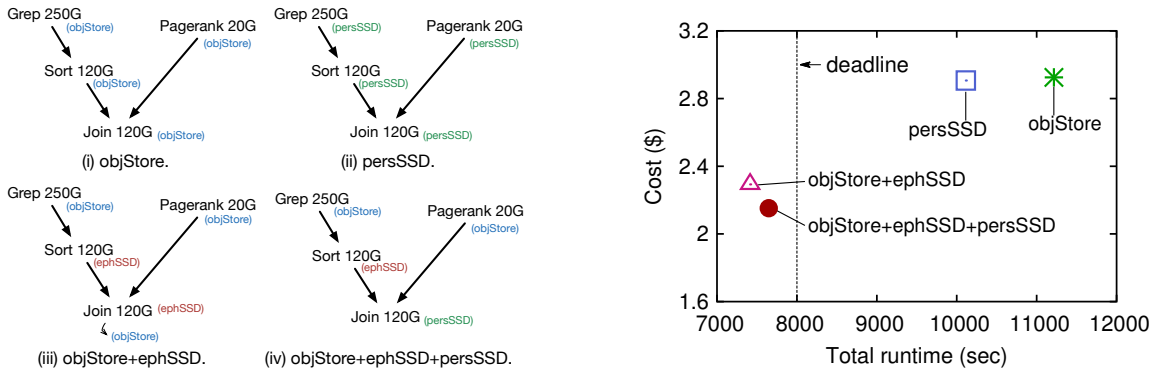
Analysis: Workload Granularity

We next study the impact of cross-job interactions within an analytics workload. While individual job-level optimization and tiering has been the major focus of a number of recent works [56, 99, 100, 131, 132, 196], we argue that this is not sufficient for data placement in the cloud for analytics workloads. To this end, we analyze two typical workload characteristics that have been reported in production workloads [49, 68, 78, 98, 169], namely data reuse across jobs, and dependency between jobs, i.e., workflows, within a workload.

Data Reuse across Jobs As reported in the analysis of production workloads from Facebook and Microsoft Bing [49, 68], both small and large jobs exhibit data re-access patterns both in the short term, i.e., input data shared by multiple jobs and reused for a few hours before becoming cold, as well as in the long term, i.e., input data reused for longer periods such as days or weeks before turning cold. Henceforth, we refer to the former as reuse-lifetime (short) and the later as reuse-lifetime (long).

To better understand how data reuse affects data placement choices, we evaluate the tenant utility of each application under different reuse patterns. Figure 4.3 shows that the choice of storage service changes based on data reuse patterns for different applications. Note that in both reuse cases we perform the same number of re-accesses (i.e., 7) over the specified time period. For instance, in `reuse-lifetime (1 week)`, data is accessed once per day, i.e., 7 accesses in a week. Similarly, data is accessed once every 8 minutes in `reuse-lifetime (1 hr)` [1]. For `ephSSD`, the input download overheads can be amortized by keeping the data in the ephemeral SSD, since the same data will be re-accessed in a very short period of time. This allows `ephSSD` to provide the highest utility for `Join` (Figure 4.3 (b)) and `Grep` (Figure 4.3 (c)) for `reuse-lifetime (1 hr)`. However, if the data is re-accessed only once per day (`reuse-lifetime (1 week)`), the cost of `ephSSD` far outweighs the benefits of avoiding input downloads. Thus, for `Sort` (Figure 4.3 (a)), `objStore` becomes the storage service of choice for `reuse-lifetime (1 week)`. For similar cost reasons, `persSSD`, which demonstrates

[1] While re-access frequency can vary for different reuse-lifetimes, we selected these two cases to highlight the changes in data placement options for the same applications due to different data reuse patterns.



(a) Four possible workflow tiering plans.

(b) Workflow tiering performance-cost trade-offs.

Figure 4.4: Possible tiering plans for a simple 4-job workflow.

the highest utility for individual applications (Figure 4.1 (b)), becomes the worst choice when long term re-accesses are considered. Furthermore, as expected, the behavior of CPU-intensive `KMeans` (Figure 4.3 (d)) remains the same across reuse patterns with highest utility achieved with `persHDD`, and the storage costs do not play a major role in its performance.

Workflows in an Analytics Workload An analytics workflow consists of a series of jobs with inter-dependencies. For our analysis, we consider the workflows where the output of one job acts as a part of an input of another job. Thus, a workflow can be abstracted as Directed Acyclic Graph (DAGs) of actions [26]. Prior research [149, 152] has shown that not only overall completion times of individual jobs in a workflow an important consideration, but meeting completion time deadlines of workflows is also critical for satisfying Service-Level Objectives (SLOs) of cloud tenants.

Consider the following example to illustrate and support the above use case. Figure 4.4(a)[2] lists four possible tiering plans for a four-job workflow. The workflow consists of four jobs and represents a typical search engine log analysis. Figure 4.4(a) (i) and Figure 4.4(a) (ii) depict cases where a single storage service, `objStore` and `persSSD`, respectively, is used for the entire workflow. As shown in Figure 4.4(b), the complex nature of the workflow not only makes these two data placement strategies perform poorly (missing a hypothetical deadline of 8,000 seconds) but also results in high costs compared to the other two hybrid storage plans. On the other hand, both the hybrid storage services meet the deadline. Here, the output of one job is pipelined to another storage service where it acts as an input for the subsequent job in the workflow. If the tenant's goal is to pick a strategy that provides the lowest execution time, then the combination `objStore+ephSSD` shown in Figure 4.4(b) provides the best result amongst the studied plans. However, if the tenant wants to choose a layout that satisfies the dual criteria of meeting the deadline and providing the lowest

[2]We do not show utility results of `Pagerank` because it exhibits the same behavior as `KMeans` in §4.2.1.

cost (among the considered plans), then the combination `objStore+ephSSD+persSSD` — that reduces the cost by 7% compared to the other tiering plan — may be a better fit.

Key Insights From this set of experiments, we infer the following. (i) Not only do data analytics workloads require use of different storage services for different applications, the data placement choices also change when data reuse effects are considered. (ii) Complex inter-job requirements in workflows necessitate thinking about use of multiple storage services, where outputs of jobs from a particular storage service may be pipelined to different storage tiers that act as inputs for the next jobs. (iii) Use of multiple criteria by the tenant, such as workflow deadlines and monetary costs, adds more dimensions to a data placement planner and requires careful thinking about tiering strategy.

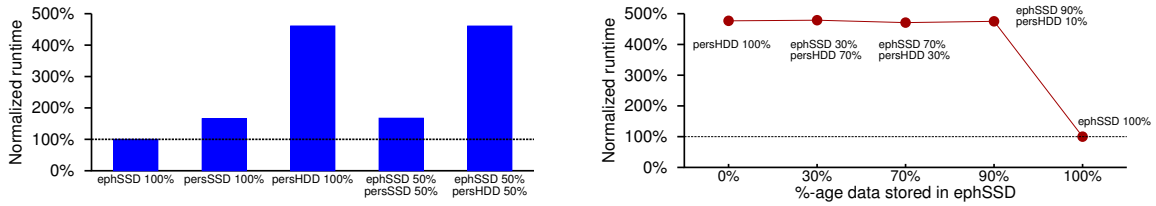
4.2.2 Shortcomings of Traditional Storage Tiering Strategies

In this following, we argue that traditional approaches to storage tiering are not adequate for being used for analytics workloads in the cloud.

Heat-based Tiering A straw man tiering approach that considers the monetary cost of different cloud storage mediums is to place hot data on `ephSSD`; semi-hot data on either `persSSD` or `persHDD`; and cold data on the cheapest `objStore`. Heat metrics can include different combinations of access frequencies, recency, etc. But the performance–cost model for cloud storage for analytics workloads is more complicated due to the following reasons. (1) The most expensive cloud storage tier (`ephSSD`) may not be the best tier for hot data, since the ephemeral SSD tier typically provides no persistence guarantee — the VMs have to persist for ensuring that all the data on ephemeral disks stays available, potentially increasing monetary costs. `ephSSD` volumes are fixed in size (Table 4.1) and only 4 volumes can be attached to a VM. Such constraints can lead to both under-provisioning (requiring more VMs to be started) and over-provisioning (wasting capacity for small datasets), in turn reducing utility. (2) Furthermore, analytics applications may derive better utility from cheaper tiers than their more expensive counterparts.

Fine-Grained Tiering Recently, hatS [132] looked at a tiered HDFS implementation that utilizes both HDDs and SSDs to improve single job performance. Such approaches focus on fine-grained tiering within a single job. While this can be useful for on-premise clusters where storage resources are relatively fixed and capacity of faster storage is limited [98], it provides few benefits for cloud-based analytics workloads where resources can be elastically provisioned. Furthermore, maintaining heat metrics at a fine-grained block or file level may be spatially prohibitive (DRAM requirements) for a big data analytics workload with growing datasets.

We contend that instead of looking at block or file-level partitioning, a more coarse-grained approach that performs job-level tiering, is needed for cloud-based analytics. To illustrate this, in Figure 4.5 we measure the performance of `Grep` under various placement configu-



(a) `ephSSD+persSSD`, `ephSSD+persHDD` are hybrid storage configurations.

(b) Fine-grained data partitioning within a job. Data is partitioned across faster `ephSSD` and slower `persHDD`.

Figure 4.5: Normalized runtime of `Grep` under different HDFS configurations. All runtime numbers are normalized to `ephSSD 100%` performance.

rations (using default Hadoop task scheduler and data placement policy) for a 6 GB input dataset requiring 24 map tasks scheduled as a single wave. As shown in Figure 4.5(a), partitioning data across a faster `ephSSD` and slower `persSSD` tier does not improve performance. The tasks on slower storage media dominate the execution time. We further vary the partitioning by increasing the fraction of input data on faster `ephSSD` (Figure 4.5(b)). We observe that even if 90% of the data is on the faster tier, the performance of the application does not improve, highlighting the need for job-level data partitioning. Such an "all-or-nothing" [49] data placement policy, i.e., placing the whole input of one job in one tier, is likely to yield good performance. This policy is not only simple to realize, but also maps well to both the characteristics of analytics workloads and elasticity of cloud storage services.

4.3 CAST Framework

We build CAST, an analytics storage tiering framework that exploits heterogeneity of both the cloud storage and analytics workloads to satisfy the various needs of cloud tenants. Furthermore, CAST++, an enhancement to CAST, provides data pattern reuse and workflow awareness based on the underlying analytics framework. Figure 4.6 shows the high-level overview of CAST operations and involves the following components. (1) The analytics *job performance estimator* module evaluates jobs execution time on different storage services using workload specifications provided by tenants. These specifications include a list of jobs, the application profiles, and the input data sizes for the jobs. The estimator combines this with compute platform information to estimate application run times on different storage services. (2) The *tiering solver* module uses the job execution estimates from the *job performance estimator* to generate a tiering plan that spans all storage tiers on the specific cloud provider available to the tenant. The objective of the solver is to satisfy the high-level tenants' goals such as achieving high utility or reducing deadline miss rates.

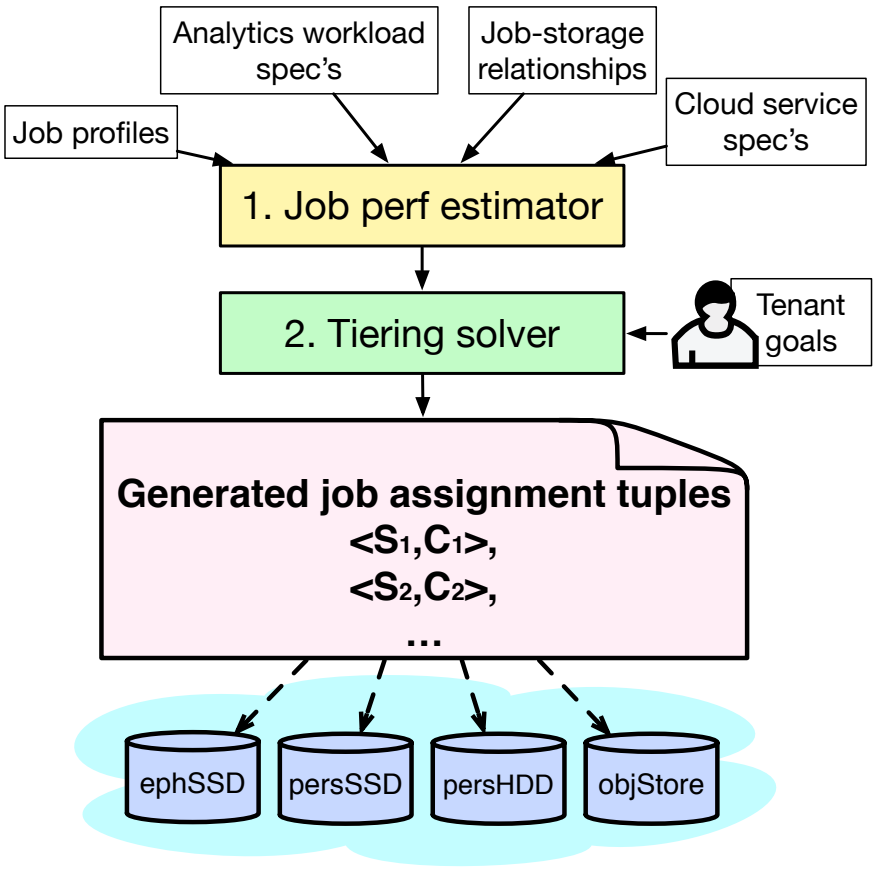


Figure 4.6: Overview of Cast tiering framework.

4.3.1 Estimating Analytics Job Performance

The well-defined execution phases of the MapReduce parallel programming paradigm [81, 119] implies that the runtime characteristics of analytics jobs can be predicted with high accuracy. Moreover, extensive recent research has focused on data analytics performance prediction [25, 99, 100, 119, 192, 193]. We leverage and adapt MRCute [119] model in Cast to predict job execution time, due to its ease-of-use, availability, and applicability to our problem domain.

Equation 4.1 defines our performance prediction model. It consists of three sub-models — one each for the map, shuffle, and reduce phases — where each phase execution time is modeled as $\#waves \times runtime\ per\ wave$. A *wave* represents the number of tasks that can be scheduled in parallel based on the number of available slots. Cast places all the data of a job on a single storage service with predictable performance, and tasks (within a job) work on equi-sized data chunks. The estimator also models wave pipelining effects — in a typical MapReduce execution flow, the three phases in the same wave are essentially serialized, but

Notation		Description
$\hat{\mathcal{R}}$	n_{vm}	number of VMs in the cluster
	m_c	number of map slots in one node
	r_c	number of reduce slots in one node
$\hat{\mathcal{M}}$	bw^f_{map}	bandwidth of a single map task on tier f
	$bw^f_{shuffle}$	bandwidth of a single shuffle task on tier f
	bw^f_{reduce}	bandwidth of a single reduce task on tier f
$\hat{\mathcal{L}}_i$	$input_i$	input data size of job i
	$inter_i$	intermediate data size of job i
	$output_i$	output data size of job i
	m	number of map tasks of job i
	r	number of reduce tasks of job i
solver	$capacity$	total capacities of different storage mediums
	$price_{vm}$	VM price ($/min)
	$price_{store}$	storage price ($/GB/hr)
	J	set of all analytics jobs in a workload
	J_w	set of all analytics jobs in a workflow w
	F	set of all storage services in the cloud
	D	set of all jobs that share the same data
	$\hat{P}$	tiering solution
decision vars	s_i	storage service used by job i
	c_i	storage capacity provisioned for job i

Table 4.3: Notations used in the analytics jobs performance prediction model and CAST tiering solver.

different waves can be overlapped. Thus, the prediction model does not sacrifice estimation accuracy. The model simplifies the predictor implementation, which is another advantage of performing coarse-grained, job-level data partitioning.

$$EST\left(\hat{\mathcal{R}}, \hat{\mathcal{M}}(s_i, \hat{\mathcal{L}}_i)\right) = \overbrace{\underbrace{\left\lceil \frac{m}{n_{vm} \cdot m_c} \right\rceil}_{\text{\# waves}} \cdot \underbrace{\left(\frac{input_i/m}{bw^{s_i}_{map}} \right)}_{\text{Runtime per wave}}}^{\text{map phase}} + \underbrace{\left\lceil \frac{r}{n_{vm} \cdot r_c} \right\rceil \cdot \left(\frac{inter_i/r}{bw^{s_i}_{shuffle}} \right)}_{\text{shuffle phase}} + \underbrace{\left\lceil \frac{r}{n_{vm} \cdot r_c} \right\rceil \cdot \left(\frac{output_i/r}{bw^{s_i}_{reduce}} \right)}_{\text{reduce phase}}. \tag{4.1}$$

The estimator $EST(.)$ predicts job performance using the information about (i) job configuration: number of map/reduce tasks, job sizes in different phases; (ii) compute configuration: number of VMs, available slots per VM; and (iii) storage services: bandwidth of tasks on a particular storage service. Table 5.1 lists the notations used in the model.

4.3.2 Basic Tiering Solver

The goal of the basic CAST tiering solver is to provide near-optimal specification that can help guide tenant's decisions about data partitioning on the available storage services for their analytics workload(s). The solver uses a simulated annealing algorithm [128] to systematically search through the solution space and find a desirable tiering plan, given the workload specification, analytics models, and tenants' goals.

CAST Solver: Modeling

The data placement and storage provisioning problem is modeled as a non-linear optimization problem that maximizes the tenant utility (U) defined in Equation 4.2:

$$max\ U = \frac{1/T}{(\$_{vm} + \$_{store})}\ , \tag{4.2}$$

$$s.t.\ c_i \geq (input_i + inter_i + output_i)\ (\forall i \in J)\ , \tag{4.3}$$

$$T = \sum_{i=1}^{J} REG(s_i, capacity[s_i], \hat{\mathcal{R}}, \hat{\mathcal{L}}_i)\ , where\ s_i \in F\ , \tag{4.4}$$

$$\$_{vm} = n_{vm} \cdot (price_{vm} \cdot T)\ , \tag{4.5}$$

$$\$_{store} = \sum_{f=1}^{F} \left(capacity[f] \cdot (price_{store}[f] \cdot \left\lceil T/60 \right\rceil)\right) \tag{4.6}$$

$$where\ \forall f \in F\ : \left\{\forall i \in J,\ s.t.\ s_i \equiv f : capacity[f] = \sum c_i\right\}.$$

The performance is modeled as the *reciprocal* of the estimated completion time in minutes ($1/T$) and the costs include both the VM and storage costs. The VM cost[3] is defined by Equation 4.5 and depends on the total completion time of the workload. The cost of each storage service is determined by the workload completion time (storage cost is charged on a hourly basis) and capacity provisioned for that service. The overall storage cost is obtained by aggregating the individual costs of each service (Equation 5.15).

Equation 4.3 defines the *capacity constraint*, which ensures that the storage capacity (c_i) provisioned for a job is sufficient to meet its requirements for all the phases (map, shuffle, reduce). We also consider intermediate data when determining aggregated capacity. For jobs, e.g., Sort, which have a selectivity factor of one, the intermediate data is of the same size as the input data. Others, such as inverted indexing, would require a large capacity for storing intermediate data as significant larger shuffle data is generated during the map phase [54]. The generic Equation 4.3 accounts for all such scenarios and guarantees that the workload will not fail. Given a specific tiering solution, the estimated total completion time of the workload is defined by Equation 5.14. Since job performance in the cloud scales with

[3]We only consider a single VM type since we focus on storage tiering. Extending the model to incorporate heterogeneous VM types is part of our future work.

capacity of some services, we use a regression model, $REG(s_i, .)$, to estimate the execution time. In every iteration of the solver, the regression function uses the storage service (s_i) assigned to a job in that iteration, the total provisioned capacity of that service for the entire workload, cluster information such as number of VMs and the estimated runtime based on Equation 4.1 as parameters. After carefully considering multiple regression models, we find that a third degree polynomial-based cubic Hermite spline [6] is a good fit for the applications and storage services considered in the paper. While we do not delve into details about the model, we show the accuracy of the splines in Figure 4.2. We also evaluate the accuracy of this regression model using a small workload in §4.4.1.

CAST Solver: Algorithms

Algorithm 3: Greedy static tiering algorithm.

Input: Job information matrix: $\hat{\mathcal{L}}$,
Output: Tiering plan $\hat{P}_{greedy}$
begin
  $\hat{P}_{greedy} \leftarrow \{\}$
  foreach *job* j *in* $\hat{\mathcal{L}}$ **do**
    $f_{best} \leftarrow f_1$ // f_1 represents the first of the
    // available storage services in F
    foreach *storage service* f_{curr} *in* F **do**
      if $Utility(j, f_{curr}) > Utility(j, f_{best})$ **then**
        $f_{best} \leftarrow f_{curr}$
    $\hat{P}_{greedy} \leftarrow \hat{P}_{greedy} \cup \{\langle j, f_{best} \rangle\}$
  return $\hat{P}_{greedy}$

Greedy Algorithm We first attempt to perform data partitioning and placement using a simple greedy algorithm (Algorithm 3). The algorithm takes the job information matrix J as the input and generates a tiering plan as follows. For each job in the workload, the utility (calculated using function $Utility(.)$) is computed using Equation 4.1 and Equation 4.2 on each storage service. The tier that offers the highest utility is assigned to the job. As astute readers will observe, while this algorithm is straightforward to reason about and implement, it does not consider the impact of placement on other jobs in the workload. Furthermore, as we greedily make placement decisions on a per-job basis, the total provisioned capacity on a tier increases. Recall that the performance of some storage services scales with capacity. Thus, the tiering decisions for some jobs (for which placement has already been done) may no longer provide maximum utility. We evaluate the impact of these localized greedy decisions in §4.4.1.

Simulated Annealing-based Algorithm In order to overcome the limitations of the Greedy approach, we devise a simulated annealing [128] based algorithm. The algorithm (Algorithm 5) takes as input workload information ($\hat{\mathcal{L}}$), compute cluster configuration ($\hat{\mathcal{R}}$), and information about performance of analytics applications on different storage services

Algorithm 4: Simulated Annealing Algorithm.

Input: Job information matrix: $\hat{\mathcal{L}}$,
Analytics job model matrix: $\hat{\mathcal{M}}$,
Runtime configuration: $\hat{\mathcal{R}}$,
Initial solution: $\hat{P}_{init}$.
Output: Tiering plan $\hat{P}_{best}$

```
begin
    P̂_best ← {}
    P̂_curr ← P̂_init
    exit ← False
    iter ← 1
    temp_curr ← temp_init
    U_curr ← Utility(M̂, L̂, P̂_init)
    while not exit do
        temp_curr ← Cooling(temp_curr)
        for next P̂_neighbor in AllNeighbors(L̂, P̂_curr) do
            if iter > iter_max then
                exit ← True
                break
            U_neighbor ← Utility(M̂, L̂, P̂_neighbor)
            P̂_best ← UpdateBest(P̂_neighbor, P̂_best)
            iter++
            if Accept(temp_curr, U_curr, U_neighbor) then
                P̂_curr ← P̂_neighbor
                U_curr ← U_neighbor
                break
    return P̂_best
```

($\hat{\mathcal{M}}$) as defined in Table 5.1. Furthermore, the algorithm uses $\hat{P}_{init}$ as the initial tiering solution that is used to specify preferred regions in the search space. For example, the results from the greedy algorithm or the characteristics of analytics applications described in Table 4.2 can be used to devise an initial placement.

The main goal of our algorithm is to find a near-optimal tiering plan for a given workload. In each iteration, we pick a randomly selected neighbor of the current solution ($AllNeighbors(.)$). If the selected neighbor yields better utility, it becomes the current best solution. Otherwise, in the function $Accept(.)$, we decide whether to move the search space towards the neighbor ($\hat{P}_{neighbor}$) or keep it around the current solution ($\hat{P}_{curr}$). This is achieved by considering the difference between the utility of the current (U_{curr}) and neighbor solutions ($U_{neighbor}$) and comparing it with a distance parameter, represented by $temp_{curr}$. In each iteration, the distance parameter is adjusted (decreased) by a $Cooling(.)$ function. This helps in making the search narrower as iterations increase; reducing the probability of missing the maximum utility in the neighborhood of the search space.

4.3.3 Enhancements: CAST++

While the basic tiering solver improves tenant utility for general workloads, it is not able to leverage certain properties of analytics workloads. To this end, we design CAST++, which enhances CAST by incorporating data reuse patterns and workflow awareness.

Enhancement 1: Data Reuse Pattern Awareness To incorporate data reuse patterns across jobs, CAST++ ensures that all jobs that share the same input dataset have the same storage service allocated to them. This is captured by Constraint 4.7 where D represents the set consisting of jobs sharing the input (partially or fully).

$$s_i \equiv s_l \ (\forall i \in D, 1 \neq l, l \in D) \tag{4.7}$$
$$where\ D = \{all\ jobs\ which\ share\ input\}$$

Thus, even though individual jobs may have different storage tier preferences, CAST++ takes a global view with the goal to maximize overall tenant utility.

Enhancement 2: Workflow Awareness Prior research has shown that analytics workflows are usually associated with a tenant-defined deadline [84, 149]. For workloads with a mix of independent and inter-dependent jobs, the basic dependency-oblivious CAST may either increase the deadline miss rate or unnecessarily increase the costs, as we show in §4.2.1. Hence, it is crucial for CAST++ to handle workflows differently. To this end, we enhance the basic solver to consider the objective of minimizing the total monetary cost (Equation 4.8) and introduce a constraint to enforce that the total estimated execution time meets the predefined deadline (Equation 4.9). This is done for optimizing each workflow separately. Each workflow is represented as a Directed Acyclic Graph (DAG) where each vertex is a job and a directed edge represents a flow from one job to another (refer to Figure 4.4(a)).

$$min\ \ \$_{total} = \$_{vm} + \$_{store}\ , \tag{4.8}$$

$$s.t.\ \ \sum_{i=1}^{J_w} REG\Big(s_i, s_{i+1}, capacity[s_i], \hat{\mathcal{R}}, \hat{\mathcal{L}}\Big) \leq deadline\ , \tag{4.9}$$

$$c_i \geq \sum_{i=1}^{J_w} \big((s_{i-1} \neq s_i) \cdot input_i + inter_i \tag{4.10}$$
$$+ (s_{i+1} \equiv s_i) \cdot output_i\big)\ , where\ s_0 = \phi\ .$$

Furthermore, Equation 4.10 restricts the capacity constraint in Equation 4.3 by incorporating inter-job dependencies. The updated approach only allocates capacity if the storage tier for the output of a job at the previous level is not the same as input storage tier of a job at the next level in the DAG. To realize this approach, we enhance Algorithm 5 by replacing the next neighbor search ($AllNeighbors(.)$) with a depth-first traversal in the workflow DAG. This allows us to reduce the deadline miss rate.

Bin	# Maps at Facebook	% Jobs at Facebook	% Data sizes at Facebook	# Maps in workload	# Jobs in workload
1				1	35
2	1—10	73%	0.1%	5	22
3				10	16
4	11—50	13%	0.9%	50	13
5	51—500	7%	4.5%	500	7
6	501—3000	4%	16.5%	1,500	4
7	> 3000	3%	78.1%	3,000	3

Table 4.4: Distribution of job sizes in Facebook traces and our synthesized workload.

4.4 Evaluation

In this section, we present the evaluation of CAST and CAST++ using a 400-core Hadoop cluster on Google Cloud. Each slave node in our testbed runs on a 16 vCPU `n1-standard-16` VM as specified in §4.2. We first evaluate the effectiveness of our approach in achieving the best tenant utility for a 100-job analytics workload with no job dependencies. Then, we examine the efficacy of CAST++ in meeting user-specified deadlines.

4.4.1 Tenant Utility Improvement

Methodology

We compare CAST against six storage configurations: four without tiering and two that employ greedy algorithm based static tiering. We generate a representative 100-job workload by sampling the input sizes from the distribution observed in production traces from a 3,000-machine Hadoop deployment at Facebook [68]. We quantize the job sizes into 7 bins as listed in Table 4.4, to enable us to compare the dataset size distribution across different bins. The largest job in the Facebook traces has 158,499 map tasks. Thus, we choose 3,000 for the highest bin in our workload to ensure that our workload demands a reasonable load but is also manageable for our 400-core cluster. More than 99% of the total data in the cluster is touched by the large jobs that belong to bin 5, 6 and 7, which incur most of the storage cost. The aggregated data size for small jobs (with number of map tasks in the range 1–10) is only 0.1% of the total data size. The runtime for small jobs is not sensitive to the choice of storage tier. Therefore, we focus on the large jobs, which have enough number of mappers and reducers to fully utilize the cluster compute capacity during execution. Since there is a moderate amount of data reuse throughout the Facebook traces, we also incorporate this into our workload by having 15% of the jobs share the same input data. We assign the four job types listed in Table 4.2 to this workload in a round-robin fashion to incorporate the different computation and I/O characteristics.

Effectiveness for General Workload

Figure 4.7 shows the results for tenant utility, performance, cost and storage capacity distribution across four different storage services. We observe in Figure 5.3(b) that CAST improves the tenant utility by 33.7% – 178% compared to the configurations with no explicit tiering, i.e., `ephSSD 100%`, `persSSD 100%`, `persHDD 100%` and `objStore 100%`. The best combination under CAST consists of 33% `ephSSD`, 31% `persSSD`, 16% `persHDD` and 20% `objStore`, as shown in Figure 4.7(c). `persSSD` achieves the highest tenant utility among the four non-tiered configurations, because `persSSD` is relatively fast and persistent. Though `ephSSD` provides the best I/O performance, it is not cost-efficient, since it uses the most expensive storage and requires `objStore` to serve as the backing store to provide data persistence, which incurs additional storage cost and also imposes data transfer overhead. This is why `ephSSD 100%` results in 14.3% longer runtime (300 minutes) compared to that under `persSSD 100%` (263 minutes) as shown in Figure 4.7(b).

The greedy algorithm cannot reach a global optimum because, at each iteration, placing a job in a particular tier can change the performance of that tier. This affects the *Utility* calculated and the selected tier for each job in all the previous iterations, but the greedy algorithm cannot update those selections to balance the trade-off between cost and performance. For completeness, we compare our approach with two versions of the greedy algorithm: `Greedy exact-fit` attempts to limit the cost by not over-provisioning extra storage space for workloads, while `Greedy over-provisioned` will assign extra storage space as needed to reduce the completion time and improve performance.

The tenant utility of `Greedy exact-fit` is as poor as `objStore 100%`. This is because `Greedy exact-fit` only allocates *just enough* storage space without considering performance scaling. `Greedy over-provisioned` is able to outperform `ephSSD 100%`, `persHDD 100%` and `objStore 100%`, but performs slightly worse than `persSSD 100%`. This is because the approach significantly over-provisions `persSSD` and `persHDD` space to improve the runtime of the jobs. The tenant utility improvement under basic CAST is 178% and 113.4%, compared to `Greedy exact-fit` and `Greedy over-provisioned`, respectively.

Effectiveness for Data Reuse

CAST$_{++}$ outperforms all other configurations and further enhances the tenant utility of basic CAST by 14.4% (Figure 5.3(b)). This is due to the following reasons. (1) CAST$_{++}$ successfully improves the tenant utility by exploiting the characteristics of jobs and underlying tiers and tuning the capacity distribution. (2) CAST$_{++}$ effectively detects data reuse across jobs to further improve the tenant utility by placing shared data in the fastest `ephSSD`, since we observe that in Figure 4.7(c) the capacity proportion under CAST$_{++}$ of `objStore` reduces by 42% and that of `ephSSD` increases by 29%, compared to CAST. This is because CAST$_{++}$ places jobs that share the data on `ephSSD` to amortize the data transfer cost from `objStore`.

Accuracy of the Regression Model

Figure 4.8 compares predicted runtime to observed runtime for a small workload consisting of 16 modest-sized jobs. The total dataset size of all jobs is 2 TB. Both the predicted and the observed runtime follow the same general trend, with an average prediction error of 7.9%, which demonstrates the accuracy of our cubic Hermite spline regression models. The margin of error is tolerable for our case, since the focus of Cast is to help tenants compare and choose among different tiering plans.

4.4.2 Meeting Workflow Deadlines

Methodology

In our next set of experiments, we evaluate the ability of Cast++ to meet workflow deadlines while minimizing cost. We compare Cast++ against four storage configurations without tiering and a fifth configuration from the basic, workflow-oblivious Cast. This experiment employs five workflows with a total of 31 analytics jobs, with the longest workflow consisting of 9 jobs. We focus on large jobs that fully utilize the test cluster's compute capacity.

We consider the completion time of a workflow to be the time between the start of its first job and the completion of its last job. The *deadline* of a workflow is a limit on this completion time, i.e., it must be less than or equal to the deadline. We set the deadline of the workflows between 15 – 40 minutes based on the job input sizes and the job types comprising each workflow. When a job of a workflow completes, its output is transferred to the input tier of the next job. The time taken for this cross-tier transfer is accounted as part of the workflow runtime by Cast++. However, since Cast is not aware of the intra-workflow job dependencies (treating all currently running workflows as a combined set of jobs), Cast cannot account for this transfer cost.

Deadline Miss Rate vs. Cost

Figure 4.9 shows the *miss rate* of workflow deadlines for the studied configurations. The miss rate of a configuration is the fraction of deadlines missed while executing the workflows using that configuration. Cast++ meets all the deadlines and incurs the lowest cost, comparable to that of `persHDD` that is the lowest-priced but the slowest tier and has a miss rate of 100%.

Cast misses 60% of the deadlines because of two reasons: (1) it selects slow tiers for several jobs in each workflow when trying to optimize for tenant utility; and (2) by not accounting for the cross-tier transfer time, it mis-predicts the workflow runtime. However, Cast incurs a lower cost compared to the non-tiered configurations, because it selects lower-priced tiers for many of the jobs.

Despite being the fastest tier, `ephSSD` misses 20% of the deadlines because of the need to fetch the input data for every workflow from `objStore`. `persSSD` misses 40% of the deadlines because it performs slightly worse than `ephSSD` for I/O intensive jobs. Finally, `objStore` misses all of the deadlines because it is slower than or as fast as `persSSD`. It incurs a higher cost because of the `persSSD`, which is needed for storing intermediate data.

In summary, CAST++ outperforms CAST as well as non-tiered storage configurations in meeting workflow deadlines, and does so while minimizing the cost of running the workflows on the cloud cluster.

4.5 Discussion

In the following, we discuss the applicability and limitations of our storage tiering solutions.

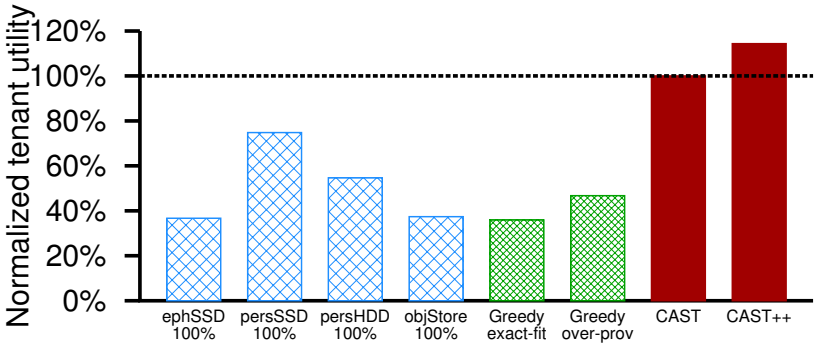
Analytics Workloads with Relatively Fixed and Stable Computations Analytics workloads are known to be fairly *stable* in terms of the number of types of applications. Recent analysis by Chen et. al. [68] shows that a typical analytics workload consists of only a small number of common computation patterns in terms of analytics job types. For example, a variety of Hadoop workloads in Cloudera have four to eight unique types of jobs. Moreover, more than 90% of all jobs in one Cloudera cluster are `Select`, `PigLatin` and `Insert` [68]. These observations imply that a relatively fixed and stable set of analytics applications (or analytics kernels) can yield enough functionality for a range of analysis goals. Thus, optimizing the system for such applications, as in CAST, can significantly impact the data analytics field.

Dynamic vs. Static Storage Tiering Big data frameworks such as Spark [204] and Impala [20] have been used for real-time interactive analytics, where dynamic storage tiering is likely to be more beneficial. In contrast, our work focuses on traditional batch processing analytics with workloads exhibiting the characteristics identified above. Dynamic tiering requires more sophisticated fine-grained task-level scheduling mechanisms to effectively avoid the straggler issue. While dynamic tiering in our problem domain can help to some extent, our current tiering model adopts a simple yet effective coarse-grained tiering approach. We believe we have provided a *first-of-its-kind* storage tiering methodology for cloud-based analytics. In the future, we plan to enhance CAST to incorporate fine-grained dynamic tiering as well.

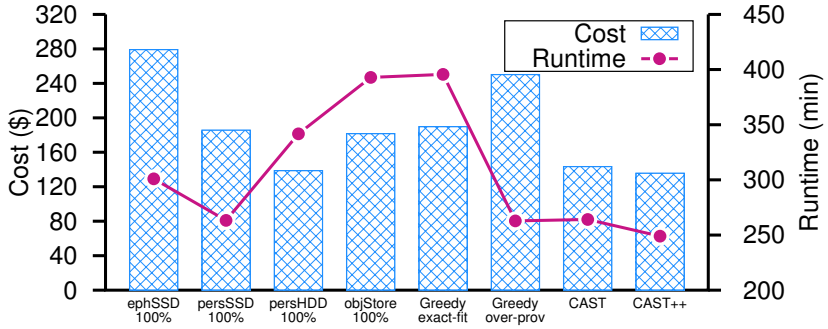
4.6 Chapter Summary

In this paper, we design CAST, a storage tiering framework that performs cloud storage allocation and data placement for analytics workloads to achieve high performance in a

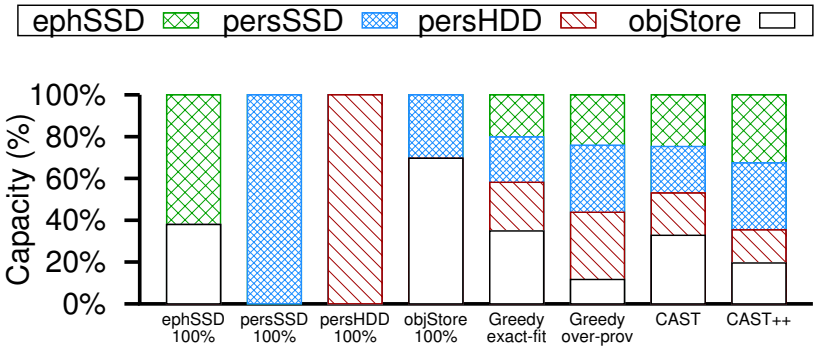
cost-effective manner. Cast leverages the performance and pricing models of cloud storage services and the heterogeneity of I/O patterns found in common analytics applications. An enhancement, Cast$_{++}$, extends these capabilities to meet deadlines for analytics workflows while minimizing the cost. Our evaluation shows that compared to extant storage-characteristic-oblivious cloud deployment strategies, Cast$_{++}$ can improve the performance by as much as 37.1% while reducing deployment costs by as much as 51.4%.



(a) Normalized tenant utility.



(b) Total monetary cost and runtime.



(c) Capacity breakdown.

Figure 4.7: Effectiveness of CAST and CAST++ on workloads with reuse, observed for key storage configurations.

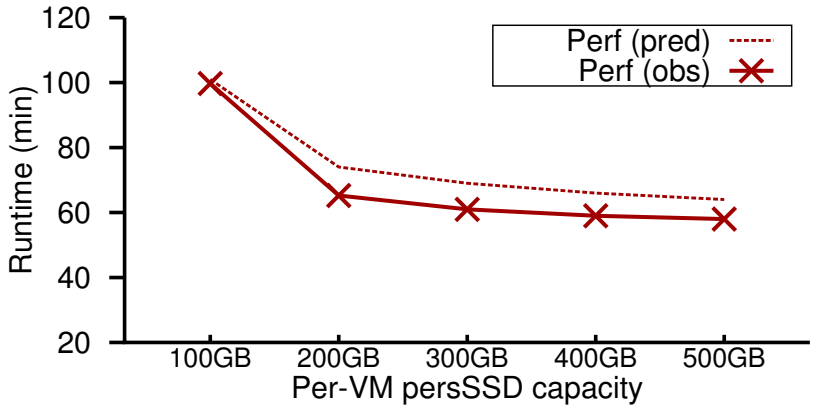


Figure 4.8: Predicted runtime achieved using CAST's performance scaling regression model vs. runtime observed in experiments by varying the per-VM `persSSD` capacity. The workload runs on the same 400-core cluster as in §4.4.1.

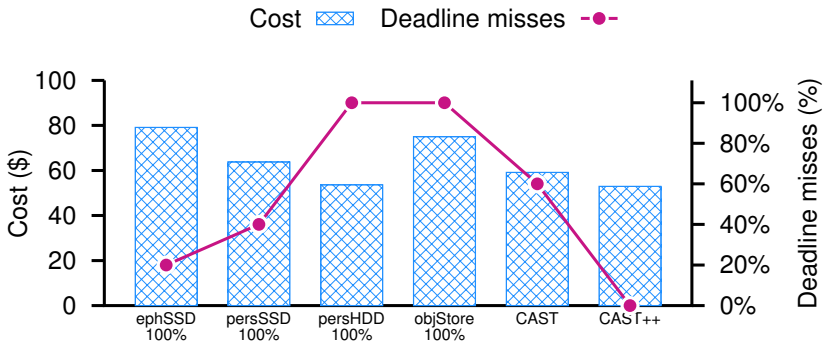


Figure 4.9: Deadline miss rate and cost of CAST$_{++}$ compared to CAST and four non-tiered configurations.

Chapter 5

Workload-aware Hybrid Cloud Object Store

From Chapter 4, we demonstrate that a carefully designed storage tiering framework can effectively improve the performance while reducing the monetary cost targeting deployed data analytics workloads for cloud tenants. One natural question that arises in the same context is, whether it is possible to reach to a win-win situation for both cloud service providers and cloud tenants, for data analytics workloads.

This chapter explores the interactions between cloud providers and tenants, and proposes a novel tiered object storage system which could effectively engage both the provider and tenants. In this work, we have attempted to reconcile the economic principles of demand and supply with the technical aspects of storage tiering through dynamic pricing to showcase the benefits for both the cloud provider as well as the tenants.

5.1 Introduction and Motivation

To make data analytics easy to deploy and elastically scale in the cloud while eliminating redundant data copying cost, cloud providers typically let their tenants run Big Data processing jobs on vast amount of data stored in object stores. For example, AWS [33], Google Cloud [13] and OpenStack [38] provide their own Hadoop to object store connectors that allow tenants to directly use object stores as a replacement of HDFS [186]. Moreover, commercial Big Data platforms such as Amazon EMR [10] and Azure HDInsight [18] go a step further and directly employ object stores as the primary storage technology.

Cloud-based object stores use low-cost HDDs as the underlying storage medium. This is because the price gap between HDDs and SSDs continue to be significant [36], especially for datacenter-scale deployments. Object stores have traditionally been used as data dumps

for large objects such as backup archives and large-volume pictures or videos; use cases where SSDs would incur a high acquisition as well as maintenance cost [171], e.g., premature device replacement. Nevertheless, recent research has shown that SSDs can deliver significant benefits for many types of Big Data analytics workloads [72, 97, 98, 132], which are thus driving the need for adopting SSDs. Newer technology on this front is promising, but does not adequately address the cost and performance trade-offs. For example, while the newer 3-bit MLC NAND technology promises to deliver higher SSD densities and potentially drive down the acquisition cost, it has taken a major toll on SSD endurance [96, 158, 198], which raises the maintenance costs.

Tiered storage is used in many contexts to balance the HDD–SSD cost and benefits by distributing the workload on a hybrid medium consisting of multiple tiers [97, 127, 151, 195]. Data analytics applications are particularly amenable to such tiered storage deployments because of the inherent heterogeneity in workload I/O patterns. The choice of tiers depends on tenants' workloads and the performance benefits achieved by using specific tiers. A growing class of data analytics workloads demonstrate different unique properties [72], which cannot be satisfied by extant heat-based tier allocation approaches [97, 127, 151, 195]. To this end, we propose an innovative tiered object store that exposes tiering control to tenants by offering the tiers under dynamic pricing. Thus, the tenants can meet their price–performance objectives by partitioning their workloads to utilize different tiers based on their application characteristics.

We argue that traditional HDD-based object stores are inefficient. (1) From the cloud tenants' perspective, an HDD-based object store cannot effectively meet their requirements (e.g., deadlines) due to the relatively slow I/O performance of HDDs. (2) From the cloud provider's perspective, an HDD-only object store does not provide any pricing leverage, which reduces profitability. A faster tier can provide a higher quality-of-service (QoS), which can be strategically priced to increase profits. Hence, a hybrid HDD–SSD approach is desirable for both cloud providers and tenants.

To verify our argument, we conducted a trace-driven simulation study by replaying two 250-job snippet traces from Facebook's Hadoop production traces [68]. We set the HDD tier price as \$0.0011/GB/day—average of the Google Cloud Storage price of \$0.00087/GB/day and the Google Cloud's HDD persistent block storage price of \$0.0013/GB/day—and the SSD tier price as \$0.0044/GB/day, i.e., 4× the HDD price. Note that we have chosen to use per-day pricing for our study as the granularity of our proposed price adjustment is **one day** (§7.4). `trace 1` consumes 12 TB data and generates 4.7 TB output, while `trace 2` consumes 18 TB data and generates 8.2 TB output. For the hybrid storage tiering case (`HDD+SSD`), the tenant places jobs in different tiers with a desired workload completion deadline of 105 hours. For this purpose, we use Algorithm 5 that essentially tries to optimize the tier allocation to meet the deadline while minimizing the cost (§5.2.2).

Figure 5.1 shows the results, from which we make three observations. (1) Workloads with `HDD-only` config. cannot meet the tenant-defined deadline and the cloud provider earns

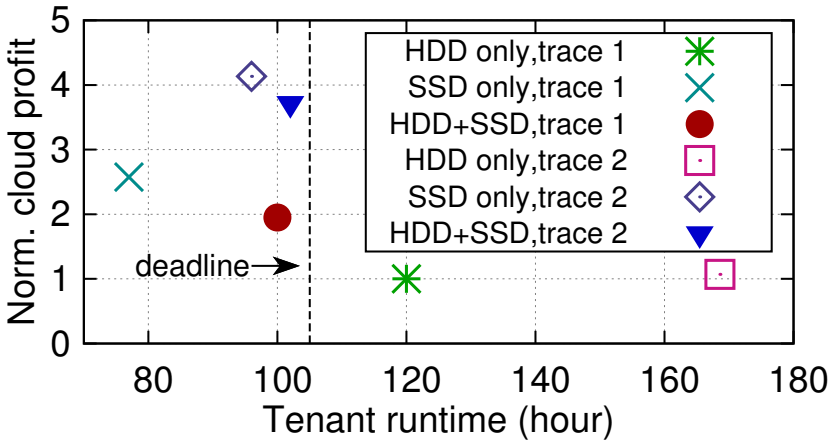


Figure 5.1: Tenant workload runtime for `trace 1` and `trace 2`, and provider's profits under different configurations.

the lowest profit. (2) With `HDD+SSD` tiering config., both workloads are able to meet the deadline, while the cloud provider sees significantly more profit (`trace 2` has larger input and output datasets and hence yields more profit). This is because the tenant places part of the workloads on the SSD tier, which is more expensive than the HDD tier. (3) `SSD only` config. improves performance, but with marginally higher profit, compared to `HDD+SSD`. This is mainly due to `HDD+SSD`'s tiering optimization. This experiment demonstrates that through object storage tiering both the cloud provider and tenants can effectively achieve their goals.

Cloud providers have a multitude of device options available for deploying object storage infrastructure, including HDDs with different RPMs, SATA and PCIe SSDs and the emerging SCM devices, etc. These options offer different performance–price trade-offs. For example, each device type offers different cost/GB and, in case of SSDs and SCMs, different endurance. In a tiered setup comprising such a variety of storage choices, estimating price points while keeping maintenance costs under control is a challenge. Moreover, *while the cloud providers encourage tenants to use more SSDs to increase their profits, they want to keep SSDs' wear-out in check* – if tenant workloads are skewed towards SSDs due to high performance requirements, providers run the risk of SSD wear-out earlier than expected, which ends up increasing management costs and decreasing overall profits.

To remedy the above issue, we introduce a dynamic pricing model that providers can leverage to mitigate additional costs and increase overall operating profits for providers. The dynamic pricing model has two objectives: (1) to balance the price-increasing and SSD wear-out rate by exploiting the trade-off between high revenue versus high operational costs (e.g., replacing SSDs) for *high profit*; (2) to provide an effective incentivizing mechanism to tenants so that the tenants can meet their goals via object store tiering in a more cost-efficient fashion. Generally, storage tiering has been looked at from just one entity's perspective. In contrast,

the novelty of this work lies in *the leader/follower game theoretic model that we adopt, where the objectives of cloud provider and tenants are either disjoint or contradictory. We take the first step towards providing a cloud-provider-driven game-theoretic pricing model through object storage tiering.* Yet another unique aspect of our storage tiering approach is *handling of the lack of information available to the players (cloud, tenants).* In extant tiering solutions adopted in private datacenters, data placement decisions are generally made by administrators who have detailed information about the systems involved. This is not true in public cloud space, where information about many aspects or details may be missing. Thus, not only the motivations different from the private deployments for providers and tenants in a public cloud, but they also have to make decisions based on partial information.

Specifically, we makes the following contributions in this paper. (1) We design a leader/follower gaming model with the goal to maximize cloud provider's profit. The provider makes the pricing decisions by estimating tenants' storage capacity demand distribution among different tiers; driven by the prices, tenants employ a simulated annealing based tiering solver to guide object storage tiering for maximizing their utility. (2) We demonstrate through trace-driven simulations that our novel object storage tiering pricing mechanism can deliver increased profit for the cloud provider and potentially achieve *win–win* for both the provider and tenants.

5.2 Model Design

We design a leader/follower cloud pricing framework with the objective of maximizing a cloud provider's profit. We model the object storage tiering for tenants and dynamic pricing for the provider, and capture the provider–tenant interactions. In our model, the game is played in two steps. First, the cloud provider (leader) makes the pricing decisions (§5.2.1) based on predictions of tenants' demand on storage resources. Second, given prices of different storage resources, a tenant (follower) makes tiering decisions based on her own requirements (§5.2.2), and the strategy is represented by the tenant's storage tiering specification, i.e., which jobs use what tier.

While the tenants can see the price changes by the provider, they are unaware of the actual reasons for the changes. Even if the tenants understood the reasons, multi-tenancy prevents modeling of the provider's behavior. Hence, in our formulation tenants can only predict the provider's price movements based on historical data. Similarly, the provider is not aware of explicit tenant requirements, and only garners information from the requested storage capacity and the writes operations (`PUT` requests are tracked for accounting purposes [2]). Thus, the provider also only uses historical information about these aspects to predict tenant demand. Consequently, both the tenants and the provider models adopted in our game are purposefully "myopic" controls for predicting only the next time slot, and not beyond that in the future.

Notation		Description
provider	F	set of all tiers in object store
	$capacity_{(n,s)}$	total capacity of tier s in time slot n
	$f_{(n,s)}$	fraction of data placed on tier s in time slot n
	$w_{(n,s)}$	writes in GB on tier s in time slot n
	$p_{(n,s)}$	price of tier s in time slot n (***decision var***)
	a	HDD class
	b	SSD class
	$\alpha_1, \beta_1, \alpha_2, \beta_2$	model parameters
	θ	parameter used to constraint SSD's price
tenant	J	set of all analytics jobs in a workload
	sz_i	dataset size of job i
	x_i	tier used by job i (***decision var***)
	c_i	capacity provisioned for job i (***decision var***)
	α_3, β_3	model parameters

Table 5.1: Notations used in the provider and tenant models.

5.2.1 Provider Model

We model the provider cost as follows. Assuming that the fraction of the cost that comes from SSDs wear-out is $t < 1$,[1] the cost can be modeled as: $cost = \frac{1}{t} \cdot \frac{p_{ssd}}{endurance} \cdot w$, where p_{ssd} is the market price of one SSD, *endurance* is the endurance lifespan of the particular SSD, and w is the amount of data written to the SSD in GB. Table 5.1 lists all the notations used in our provider and tenant models. The pricing decision making process can be modeled as a non-linear optimization problem that maximizes the profit defined by Equation 5.1.

$$max \quad profit = \sum_i \Big(\sum_f (capacity_f \cdot f_{(i,f)} \cdot p_{(i,f)}) - cost_i \Big) \tag{5.1}$$

$$s.t. \quad f'_{(n+1,b)} = \alpha_1 \cdot f_{(n,b)} - \beta_1 \cdot (p'_{(n+1,b)} - p_{(n,b)}) \tag{5.2}$$

$$f'_{(n+1,a)} = 1 - f'_{(n+1,b)} \tag{5.3}$$

$$f_{(n,b)} = \alpha_1 \cdot f_{(n-1,b)} - \beta_1 \cdot (p_{(n,b)} - p_{(n-1,b)}) \tag{5.4}$$

$$f_{(n,a)} = 1 - f_{(n,b)} \tag{5.5}$$

$$w'_{(n+1,b)} = \alpha_2 \cdot w_{(n,b)} + \beta_2 \cdot (f'_{(n+1,b)} - f_{(n,b)}) \tag{5.6}$$

$$w_{(n,b)} = \alpha_2 \cdot w_{(n-1,b)} + \beta_2 \cdot (f_{(n,b)} - f_{(n-1,b)}) \tag{5.7}$$

$$\forall i: \ w_{(i,b)} \leq \mathcal{L}_i \tag{5.8}$$

$$\forall i, s: \ 0 \leq capacity_i \cdot f_{(i,s)} \leq L_s \tag{5.9}$$

$$\forall i: \ p_{min,b} \leq p_{(i,b)} \leq \theta \cdot p_{(i,a)}, \ \ where \ \ \theta > 1 \tag{5.10}$$

$$\forall i, s: \ f_{(i,s)} \leq 1 \ where \quad i \in \{n, n+1\}, \ \ s \in F \tag{5.11}$$

[1] We choose to use a fixed t for simplicity; in real world, there are numerous factors that come into play and t may not be a constant.

In a time slot n, we predict the SSD demand proportion for the next time slot $n+1$ with Equation 5.2, which depends on the difference between the predicted SSD price for $n+1$ and the calculated SSD price for n. The predicted HDD demand proportion is determined by Equation 5.3. Similarly, Equation 5.4 and 5.5 define the predicted SSD and HDD demand proportion for n, respectively, and Equation 5.11 enforces the proportion range.

Equation 5.6 predicts the amount of data that will be written to SSDs, which is determined by the difference of predicted SSD demand proportion in time slot $n+1$ to that in time slot n. If the SSD demand is predicted to increase, it implies that the amount of data that will be absorbed by the SSD tier will also increase. Equation 5.8 defines the SSD tier data writing constraint, which ensures that the expected amount of data written to the SSD tier will not exceed the threshold that is calculated based on accumulated historical statistics. The factor indirectly controls the value adaptation of decision variables $p_{(n,b)}$ and $p'_{(n+1,b)}$. The storage capacity limit in the cloud datacenter is defined by Equation 5.9. We assume HDD prices $p_{(n,a)}$ and $p_{(n+1,a)}$ are fixed, and SSD prices are constrained in a range given by Equation 5.10.[2]

5.2.2 Tenant Model

The data placement and storage provisioning at the tenant side is modeled as a non-linear optimization problem as well. The goal is to maximize *tenant utility* as defined by Equation 5.12.

$$max \quad utility = \frac{1}{(T \cdot \$)} \tag{5.12}$$

$$s.t. \quad \forall i \in J: \ c_i \geq sz_i \tag{5.13}$$

$$T = \sum_{i=1}^{J} (x_i, c[s_i], \hat{\mathcal{R}}, \hat{\mathcal{L}}_i) + penalty(\text{data migrated})$$

$$\leq deadline\ , \quad where\ x_i \in F \tag{5.14}$$

$$\$ = \sum_{s=1}^{F} \left(c[s] \cdot (p_{(n,s)} \cdot \left\lceil {}^{T}/_{60} \right\rceil) \right) \tag{5.15}$$

$$where\ \forall s \in F, \left\{ \forall i \in J,\ s.t.\ x_i \equiv f : c[s] = \sum c_i \right\},$$

$$p'_{(n+1,b)} = \alpha_3 \cdot p_{(n,b)} + \beta_3 \cdot p_{(n-1,b)} \tag{5.16}$$

The performance of the tenant's workload is modeled as the *reciprocal* of the estimated completion time in minutes ($^1/_T$) and the cost includes mainly the storage costs. The cost of each storage service is determined by the workload completion time (storage cost is charged on a hourly basis) and capacity provisioned for that service. The overall storage cost is

[2]We plan to include IOPS per client in our future pricing models.

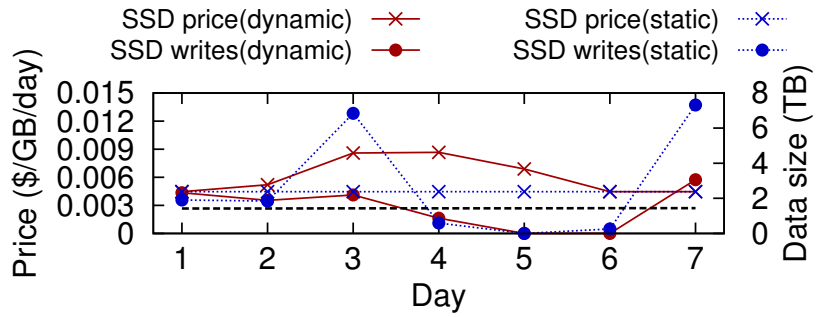
Algorithm 5: Tiering solver.

Input: Job information matrix: $\hat{\mathcal{L}}$, Analytics job model matrix: $\hat{\mathcal{M}}$, Runtime configuration: $\hat{\mathcal{R}}$, Initial solution: $\hat{P}_{init}$.
Output: Tiering plan $\hat{P}_{best}$

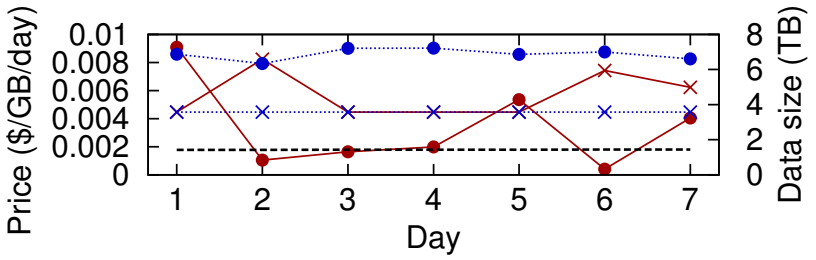
```
begin
    P̂_best ← {}
    P̂_curr ← P̂_init
    exit ← False
    iter ← 1
    temp_curr ← temp_init
    U_curr ← Utility(M̂, L̂, P̂_init)
    while not exit do
        temp_curr ← Cooling(temp_curr)
        for next P̂_neighbor in AllNeighbors(L̂, P̂_curr) do
            if iter > iter_max then
                exit ← True
                break
            U_neighbor ← Utility(M̂, L̂, P̂_neighbor)
            P̂_best ← UpdateBest(P̂_neighbor, P̂_best)
            iter++
            if Accept(temp_curr, U_curr, U_neighbor) then
                P̂_curr ← P̂_neighbor
                U_curr ← U_neighbor
                break
    return P̂_best
```

obtained by aggregating the individual costs of each tier in the object store (Equation 5.15). Equation 5.13 defines the *capacity constraint*, which ensures that the storage capacity (c_i) provisioned for a job is sufficient to meet its requirements for all the workload phases (map, shuffle, reduce). Given a specific tiering solution, the estimated total completion time of the workload is defined by Equation 5.14, and constrained by a tenant-defined *deadline*. Equation 5.16 is the price predictor at the tenant side. The predicted price value can also be supplied as a hint by the cloud provider. The function *penalty*(.) serves as a penalty term that the tenant takes into account in terms of performance loss (e.g., longer completion time) while deciding tiers.

We devise a simulated annealing based algorithm [72] (Algorithm 5) for computing tenants' data partitioning and job placement plans. The algorithm takes as input workload information ($\hat{\mathcal{L}}$), compute cluster configuration ($\hat{\mathcal{R}}$), and information about performance of analytics applications on different storage services ($\hat{\mathcal{M}}$) as defined in Table 5.1. $\hat{P}_{init}$ serves as the initial tiering solution that is used to specify preferred regions in the search space. The results from a simple greedy algorithm based on the characteristics of analytics applications (e.g., the four described in §7.4) can be used to devise an initial placement.



(a) A 7-day trace.



(b) A single-day trace repeated 7 times.

Figure 5.2: Dynamic pricing by the provider and the tenant's response for two 7-day workloads.

5.3 Evaluation

We have used trace-driven simulations to demonstrate how our cloud–tenant interaction models perform in practice. We use the production traces collected from a 3,000-machine Hadoop deployment at Facebook [68]. The original traces consist of 25,428 Hadoop jobs; we chose to use a snippet of 1,750 jobs to simulate a 7-day workload. The workload runs on a cloud object store with built-in tiering mechanism that tenants can control. We set the time slot for our models to one day.

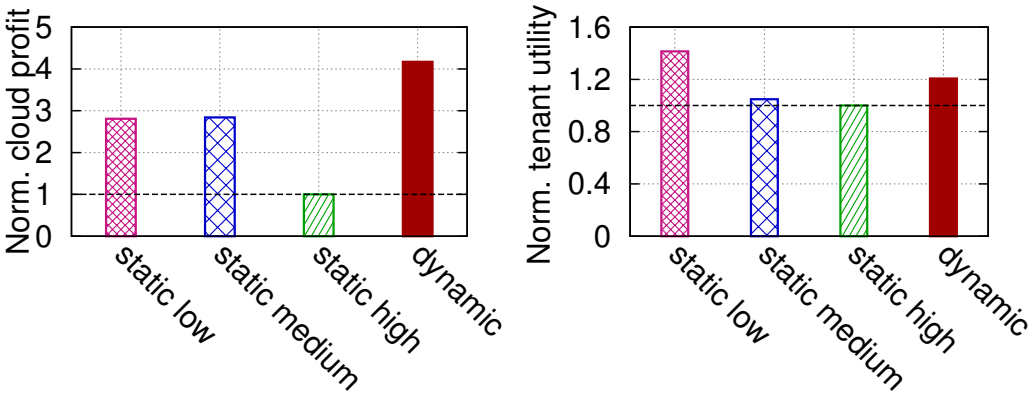
We assign to our workload, in a round-robin fashion, four analytics applications that are typical components of real-world analytics [68, 205] and exhibit diversified I/O and computation characteristics. `Sort`, `Join` and `Grep` are I/O-intensive applications. The execution time of `Sort` is dominated by the shuffle phase I/O. `Grep` spends most of its runtime in the map phase I/O, reading input and finding records that match given patterns. `Join` represents an analytic query that combines rows from multiple tables and performs the join operation during the reduce phase, making it reduce intensive. `KMeans` is an iterative CPU-intensive clustering application that expends most of its time in the compute phases of map and reduce iterations.

Figure 5.2 shows price variation by the cloud provider's model based on the amount of data

written by the tenant to the SSD tier on a per-day basis. The HDD price is fixed, while the SSD price is dynamically adjusted on a daily basis for `dynamic` pricing (the pricing is the same as for Figure 5.1). Under `static` pricing, the provider sets a static price and tenants periodically run Algorithm 5, whereas under `dynamic` pricing, the provider and tenants interact. The per-day write limit $\mathcal{L}_n$ is dynamically adjusted based on the amount of writes from the tenant side (though not discernible in the figure). Figure 5.2(a) shows the price changes for a 7-day trace, with a different workload running on each day. We observe that as the amount of writes by the tenant on SSD tier increases above the write limit, the cloud provider begins to adjust the SSD price. The tenant's model will adjust the tiering strategy to either put more data in the HDD tier or pay more for the SSD tier in the case of a strict deadline requirement. Since, each day has a different workload and hence a different deadline. For example, from day 4, the tenant, with the goal of maximizing the tenant utility, allocates fewer jobs to the SSD tier. Once the SSD writes are reduced below the threshold, the provider lowers the SSD tier price to incentivize the tenant to use more of the SSD resource on the next day. The tenant responds to this change on day 7, where the workload deadline is stricter than the previous 2 days.

Figure 5.2(b) shows the price changes for a 7-day period with the same single-day trace replayed every day. This trace shows stronger correlation between per-day SSD writes from the tenant and the SSD price from the provider. This workload exhibits the same specifications every day (e.g. dataset size, a relaxed deadline, etc.), thus the daily writes on the SSD tier remain stable under static pricing. However, the dynamic pricing model can effectively respond to the spike in the amount of writes on the first day and adjust the SSD price accordingly. Given the increased SSD price, the tenant tries to reduce their monetary cost by migrating more jobs to the cheaper HDD tier, while still meeting the deadline. When the provider lowers the SSD price in response, the tenant increases their use of SSD, prompting the provider to increase the SSD price again. This interaction of the tenant and the provider results in an average of 2.7 TB/day SSD writes compared to an average of 7 TB/day under static pricing (with 0% deadline miss rate for both cases). The test demonstrates that our dynamic pricing model can adaptively adjust the SSD pricing based on the amount of data written to the SSD tier to maintain high profits while keeping the SSD wear-out under control by keeping write loads to SSD in check.

In our next test, we examine the impact of different SSD pricing models on provider profit and tenant utility, i.e., the cloud–tenant interaction. Figure 5.3 shows the results. We choose three static prices for SSDs: low (the minimum SSD price that we use: \$0.0035/GB/day), medium (\$0.0082/GB/day), and high (the maximum SSD price, \$0.0121/GB/day). We also compare static prices with our dynamic pricing model. As observed in Figure 5.3(a), dynamic pricing yields the highest provider profit as it increases the price based on SSD writes from the tenant. Both `static low` and `medium high` yield similar profits that are 32.6% lower than that gained under dynamic pricing. This is because `static medium` results in more jobs placed on the HDD tier, which lowers the tenant cost while causing longer workload completion time. `static low` is not able to generate enough profit due to the very



(a) Cloud profit. (b) Tenant utility.

Figure 5.3: Cloud profit and tenant utility averaged over 7 days.

low price, while under `static high` the tenant solely migrates all the jobs to the HDD tier, thus resulting in low profit.

Next, we examine the tenant utility in Figure 5.3(b). With `static low` SSD price, the tenant utility is 17.1% higher than that achieved under dynamic pricing. However, this would result in significantly shortened SSD lifetime (by as much as 76.8%), hence hurting the cloud profit in the long term. With `static high` SSD price, the tenant utility is reduced by 17.1% compared to that of dynamic pricing, as the tenant shifts most jobs to the HDD tier. `static medium` SSD price yields slightly higher tenant utility as compared to `static high` but still 13.1% lower than that seen under dynamic pricing. This is because the tenant has to assign some jobs to the faster SSD tier in order to guarantee that the workload does not run for too long. Dynamic pricing, on the other hand, maintains the tenant utility at a reasonably high level (higher than both `static medium` and `static high` but slightly lower than `static low`), while guaranteeing that the SSD lifetime constraints are met. This demonstrates that our dynamic pricing model can effectively achieve a win–win for both the cloud provider and tenants.

5.4 Discussion

While a lot of research has looked at the technical aspects of building clouds, their impact and their use by the tenants, we believe the current cloud pricing mechanism (especially for storage services) is vague and lacks transparency.

We believe that our paper will lead to discussion on the following points of interest: (1) The need for revisiting pricing strategies in established hybrid storage deployments and practices. Since storage tiering is a well-studied area, we believe that the paper will lead to hot discus-

sion on how the paradigm shift is happening, and why the extant approaches and ideas need to be revisited. A particular point of interest in the context of tiering is that the objectives of different players are often times in conflict. (2) Issues such as how useful can our pricing "knob" be for the cloud provider to shape tenant behavior to suit the provider requirements, and at what granularity should the proposed price variations be implemented, are part of our future work and would make for a productive discussion. (3) The role of flash and other emerging technologies in cloud-based object stores.

An unexplored issue is tenant behavior modeling. Our preliminary results assume a smart tenant using models. However, in reality, tenants can have very different behaviors and utility functions. Furthermore, we have not looked at multi-tenancy in detail. For instance, a "naive" or "rogue" tenant that performs a lot of SSD writes without considering their utility can cause the cloud to increase the price of SSDs, thus affecting the utility of other well-behaved tenants. Another open aspect of our current work is investigating the game theoretic results of our model. These include the behavior of provider's profit and the tenant's utility when the system reaches equilibrium and the comparison of these objectives under Nash and Stackelberg equilibria.

Furthermore, prices of SSDs are falling. Even though the gap between HDD and SSD prices is still wide today, in the future with increase in NAND flash production, improvement in flash yields, lithographic improvements such as 3D stacking, etc., can reduce the price difference, resulting in SSDs becoming the de facto storage medium in cloud environments. From the tenant side, while researchers have shown the merit of using SSDs (e.g., in interactive HBase workloads [98]), their use in batch-oriented analytics workloads is just starting. Indeed, in another research work [72], we have shown through real experiments on Google Cloud that SSDs can provide great benefits especially when considering heterogeneity in cloud storage services, enforcing our belief that our future hybrid object store prototyping efforts will yield desirable win–win solutions.

5.5 Chapter Summary

We show that by combining dynamic pricing with cloud object storage tiering, cloud providers can increase their profits while meeting the SSD wear-out requirements, and tenants can effectively achieve their goals with a reasonably high utility. We demonstrate this win–win situation via real-world trace-drive simulations. In our future work, we plan to explore different tiering algorithms and best dynamic pricing models in multi-tenant clouds.

Chapter 6

Workload-aware Endurance-optimized Flash Caching

While Chapter 3, Chapter 4, and Chapter 5 present various approaches for solving the data management issues that exist in distributed and cloud storage services, this chapter looks into the data management problems that persist in all flash-boosted local filesystem hierarchy. The work included in this chapter has the potential to benefit many different types of high-level storage and data-intensive applications such as relational database systems, key-value stores, and distributed filesystems.

6.1 Introduction

SSDs typically provide a flash translation layer (FTL) within the device, which maps from logical block numbers to physical locations. A host can access individual file blocks that are kilobytes in size, and if some live blocks are physically located in the same erasure unit as data that can be recycled, the FTL will garbage collect (GC) by copying the live data and then erasing the previous location to make it available for new writes.

As an alternative to performing GC in the FTL, the host can group file blocks to match the erasure unit (also called *blocks* in flash terminology). While some research literature refers to these groupings as "blocks" (e.g., RIPQ [188]), there are many other names for it: write-evict unit [144], write unit [176], erase group unit [174], and *container* in our own recent work on online flash cache replacement [146]. Thus we use "container" to describe these groupings henceforth.

Containers are written in bulk, thus the FTL never sees partially dead containers it needs to GC. However, the host must do its own GC to salvage important data from containers before reusing them. The argument behind moving the functionality from the SSD into the

host is that the host has better overall knowledge and can use the SSD more efficiently than the FTL [135].

Flash storage can be used for various purposes, including running a standalone file system [82, 121, 134, 163, 201] and acting as a cache for a larger disk-based file system [86, 89, 101, 130, 173, 179, 182, 202]. The latter is a somewhat more challenging problem than running a file system, because a cache has an additional degree of freedom: data can be stored in the cache or bypassed [112, 181], but a file system must store all data. In addition, the cache may have different goals for the flash storage: maximize hit rate, regardless of the effect on flash endurance; limit flash wear-out, and maximize the hit rate subject to that limit; or optimize for some other utility function that takes both performance and endurance into account [146].

Flash cache replacement solutions such as RIPQ [188] and Pannier [146] consider practical approaches given a historical sequence of file operations; i.e., they are "online" algorithms. Traditionally, researchers compare online algorithms against an offline optimal "best case" baseline to assess how much room an algorithm might have for improvement [63, 89, 93, 208]. For cache replacement, Belady's MIN algorithm [57] has long been used as that baseline.

Figure 5.1 plots read hit ratio (RHR) against the EPBPD required by a given algorithm assuming SSD sizes of 1% or 10% of the working set size (WSS) of a collection of traces (described in §6.4.1). MIN achieves an average RHR improvement of 10%–75% compared to LRU, a widely used online algorithm. Using future knowledge makes a huge difference to RHR: Pannier covers only 33% to 52% of the gap from LRU to MIN, while RIPQ+ (described in §6.4.3) sees about the same RHR as LRU. However, MIN is suboptimal when considering not only RHRs but also flash endurance: it will insert data into a cache if it will be accessed before something currently in the cache, even if the new data will itself be evicted before being accessed. Such an insertion increases the number of writes, and therefore the number of erasures, without improving RHR; we refer to these as *wasted writes*. In this paper we explore the tradeoffs between RHR and erasures when future accesses are known.

A second complicating factor arises in the context of flash storage: containers. Not only should an offline algorithm not insert unless it improves the RHR, it must be aware of the container layout. For example, data that will be invalidated around the same time would benefit by being placed in the same container, so the container can be erased without the need to copy blocks.

We believe that an offline *optimal* flash replacement algorithm not only requires future knowledge but is computationally infeasible. We develop a series of heuristics that use future knowledge to make best-effort cache replacement decisions, and we compare these heuristics to each other and to online algorithms. Our *container-optimized offline heuristic* maintains the same RHR as MIN. The heuristic identifies a block that is inserted into the cache but evicted before being read, and omits that insertion in order to avoid needing to erase the region where that block is stored. At the same time, the heuristic consolidates blocks that will be evicted at around the same time into the same container when possible, to minimize

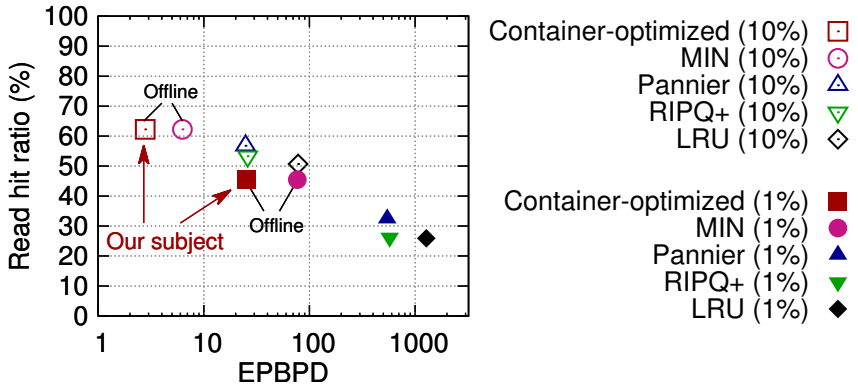


Figure 6.1: A RHR vs. endurance (EPBPD, on a log scale) scatter-plot of results for different caching algorithms. We report the average RHR and EPBPD across a set of traces, using cache sizes of 1% or 10% of the WSS. (Descriptions of the datasets and the full box-and-whisker plots appear below.) RIPQ+ and Pannier are online algorithms, described in §6.4.3. The goal of our container-optimized offline heuristic is to reduce EPBPD with the same RHR as MIN, an offline algorithm that provides the optimal RHR without considering erasures.

GC activities. One important finding is that simple techniques (e.g., omitting insertions and GC copies that are never reread) provide most of the benefit; the more complicated ones (e.g., consolidating blocks that die together) have a marginal impact on saving erasures at relatively large cache sizes. Alternatively, we describe other approaches to maximize RHR subject to erasure constraints. Figure 5.1 provides examples in which the average RHR is the same maximum achievable by MIN, while lowering EPBPD by 56%–67%. Interestingly, the container-based online algorithms reduce EPBPD relative to LRU by similar factors.

Specifically, we make the following contributions:

- We thoroughly investigate the problem space of offline compound object caching in flash.
- We identify and evaluate a set of techniques to make offline flash caching container-optimized.
- We present a multi-stage heuristic that approximates the offline optimal algorithm; our heuristic can serve as a useful approximate baseline for analyzing any online flash caching algorithm.
- We experiment with our heuristic on a wide range of traces collected from production/deployed systems, to validate that it can provide a practical upper bound for both RHR and lifespan.

6.2 Quest for the Offline Optimal

A flash device contains a number of *flash blocks*, the unit of erasures, referred to in our paper as *containers* to avoid confusion with file blocks. But many of the issues surrounding flash caching arise even in the absence of containers that aggregate multiple file blocks. We refer to the case where each file block can be erased individually as **unit caching**, and we describe the metrics (§6.2.1) and algorithms (§6.2.2) in that context. This separates the general problem of deciding what to write into the flash cache in the first place from the overhead of garbage collecting containers; we return to the impact of containers in §6.2.3. In §6.2.2–6.2.3 we also introduce a set of techniques for eliminating wasted writes. We then discuss how to handle user-level writes in §6.2.4. Finally, we summarize the algorithms of interest in §6.2.5.

6.2.1 Metrics

The principal metrics of concern are:

Read Hit Ratio (RHR): The ratio of read I/O requests satisfied by the cache (DRAM cache + flash cache) over total read requests.

The Number of Flash Erasures: In order to compare the impact on lifespan across different algorithms and workloads, we focus on the EPBPD required to run a given algorithm on a given workload and cache size. The total number of erasures is the product of EPBPD, capacity, and workload duration.

Flash Usage Effectiveness (FUE): The FUE metric [146] endeavors to balance RHR and erasures. It is defined as the number of bytes of flash hit reads divided by flash writes, including client writes and internal copy-forward (CF) writes. A score of 1 means that, on average, every byte written to flash is read once, so higher scores are better. It can serve as a utility function to evaluate different algorithms. We define *Weighted FUE* (**WFUE**), a variant of FUE that considers both RHR and erasures and uses a weight to specify their relative importance:

$$\text{WFUE} = \alpha * (\text{RHR}_A/\text{RHR}_M) + (1-\alpha) * (E_M - E_A)/E_M$$

The utility of an algorithm is determined by comparing the RHR and erasures (E) incurred by the algorithm, denoted by A, to the values for M^+ (an improved MIN, described in §6.2.2, denoted here by M for simplicity).[1] If α is low and an algorithm saves many writes in exchange for a small reduction in RHR, WFUE will increase.

[1] Though these two factors have different ranges and respond differently to changes, WFUE controls the value of both via normalization so that the higher each factor yields, the better the algorithm performs with respect to that goal. A negative value due to high erasures would demonstrate the deficiency of the algorithm in saving erasures. Hence, WFUE can serve as a general metric for quantitatively comparing different heuristics.

6.2.2 Objectives and Algorithms

Depending on the goals of end users, we may have different objective functions. Optimizing for RHR irrespective of erasures is trivial: the performance metric RHR serves as a naïve but straightforward goal for which MIN can easily get an optimal solution, without considering the flash endurance. However, MIN serves as a baseline against which to compare other algorithms. Taking erasures into account, we identify three objectives of interest. We describe each briefly to set the context for comparison, then elaborate on heuristics to optimize for them. (We do not claim their optimality, leaving such analysis to future work.)

O1: Maximal RHR The purpose of objective **O1** is to *minimize erasures subject to maximal RHR*. If we consider the RHR obtained by MIN, there should be a sequence of cache operations that will preserve MIN's hit ratio while reducing the number of erasures. Belady's **MIN** caches any block that either fits in the cache, or which will be reaccessed sooner than some other block in the cache. It does not take into account whether the block it inserts will itself be evicted from the cache before it is accessed.

The first step to reducing erasures while keeping the maximal RHR is to identify *wasted cache writes* due to eviction. **Algorithm M+** is a variant of MIN that identifies which blocks are added (via reads or writes) to the cache and subsequently evicted without rereference, then no longer inserts them into the cache (R_N in Table 6.1, where $N = 1$).

Technique	Description	C
R_N	omit insertions reread $< N$ times	✗
TRIM	notify GC to omit dead blocks	✓
CFR	avoid wasted CF blocks	✓
E	segregate blocks by evict time	✓

Table 6.1: Summary of offline heuristic techniques used for eliminating wasted writes to the flash cache. C: container-optimized.

It is unintuitive, but cache writes can be wasted even if they result in a read hit. As an example, assume block A is in cache at time t_0, and will next be accessed at time $T > t_0$. If block B is accessed at t_0, and will be accessed exactly one more time at time $T - 1$, MIN dictates that A be replaced by B. However, by removing B, there is still one miss (on B rather than A), while an extra write has occurred. Leaving A in the cache would have the same RHR but one fewer write into the cache.

Ultimately, our goal is to identify a Pareto-optimal solution set where it is impossible to reduce the number of erasures without reducing RHR. This requires that no block be inserted if it does not improve RHR, but the complexity of considering every insertion decision in that light is daunting. Thus we start with eliminating cache insertions that are completely wasted and leave additional trades of one miss against another to future work.

An offline heuristic **Algorithm H** that approximates **M+** works as follows:

Step 1 Annotate each entry with its next reference.

Step 2 Run **MIN** to annotate the trace with a sequence of cache insertions and evictions, given a cache capacity. Note all insertions that result in being evicted without at least one successful reference.

Step 3 Replay the annotated trace: do not cache a block that had not been accessed before eviction.

O2: Limited Erasures In some cases a user will be willing to sacrifice RHR in order to reduce the number of erasures. In fact, given limits on the total number of erasures of a given region of flash, it may be essential to make that tradeoff. Thus, **O2** first limits erasures to a particular rate, such as 5 EPBPD. (The EPBPD rate is multiplied by the size of the flash cache and the duration of the original trace to compute a total budget for erasures.) *Given an erasure limit, the goal of **O2** is to maximize RHR.* Note that the rate of erasures is averaged across an entire workload, meaning that the real limit is the total number of erasures; EPBPD is a way to normalize that count across workloads or configurations.

We can modify **Algorithm H** for **O2** to have a threshold. $\mathbf{H}_T$ works as follows:

Step 1 Run **H** and record all insertions. Annotate each record with the number of read hits absorbed as a result of that insertion, and count the total number of insertions resulting in a given number of read hits.

Step 2 Compute the number of cache insertions I performed in the run of **H** and the number of insertions I' allowed to achieve the EPBPD threshold T. If $I > I'$ then count the cumulative insertions CI resulting in 1 read hit, 2 read hits, and so on until $I - CI = I'$. Identify the reuse count, R, at which eliminating these low-reuse insertions brings the total EPBPD to the threshold T. Call the number of cache insertions with R reuses that must also be eliminated the leftover, L.

Step 3 Rerun **H**, skipping all insertions resulting in fewer than R read hits, and skipping the first L insertions resulting in exactly R hits.

Algebraically, we can view the above description as follows: Let A_i represent the count of cache insertions absorbing i hits.

$$I = \sum_{i=1}^{n} A_i \qquad \text{and} \qquad CI = (\sum_{i=1}^{R-1} A_i) + L$$

We identify R such that this results in $I - CI = I'$.

O3: Maximize WFUE The goal of **O3** is to *maximize WFUE*, which combines RHR and erasures into a single score to simplify the comparison of techniques (§6.2.1). Intuitively, the user may want to get the highest read hits per erasure (i.e., best "bang-for-the-buck" considering the user pays the cost of device endurance for RHR).

To compare the tradeoffs between RHR and erasures, we consider a variant of **H**, **Algorithm $\mathbf{H}_N$**, which omits cache insertions that are reread $< N$ times (R_N in Table 6.1). This is similar to the threshold-based **Algorithm $\mathbf{H}_T$**, but the decision about the number of reaccesses necessary to justify a cache insertion is *static*. An increase in the minimum number of reads per cache insertion should translate directly to a higher FUE, though the writes due to GC are also a factor. For WFUE, the value of α determines whether such a threshold is beneficial.

6.2.3 Impact from Containers

The metrics and objectives described in §6.2.1–6.2.2 apply to the unit caching scenario, in which each block may be erased separately, but they also apply to the container environment. The aim is still to minimize erasures subject to a maximal RHR, to maximize RHR subject to a limit on erasures, or to maximize a utility function of the two.

However, the approach to *solving* the optimization problem varies when containers are considered. This complexity arises because there is an extra degree of freedom: not only does an algorithm need to decide *whether* to cache a block, it must decide *where* it goes and whether to reinsert it during GC. Regarding placement, one option is to cache data in a container-oblivious manner. For instance, a host could write each block to a distinct location in flash and rely on an FTL to reorganize data to consolidate live data and reuse areas of dead data. This might result in a significant overhead from the FTL unwittingly copying forward blocks that MIN knows are no longer needed, so adding the SSL *TRIM* [190] command to inform the FTL that a block is dead can reduce erasures significantly. As shown in Table 6.1, we categorize TRIM as *container-optimized*, because an FTL itself manages data at the granularity of containers.

For CF, the first step is to supplement the annotations from §6.2.2 with information about blocks that are CF and not reaccessed. Copy-Forward Reduction (*CFR* in Table 6.1) effectively extends TRIM with the logic of R_1, by identifying "wasted" CFs; however, eliminating *all* needless CFs is difficult. With the smallest cache, on average this reduces the erasures due to wasted CFs from 4% to 1%; repeating this step a few more times brings it down another order of magnitude but does not completely eliminate wasted CFs. This is because (for a small cache) there is always a block to CF that has not yet been verified as a useful copy nor marked as wasted. Note that while it seems appealing to simply not copy something forward that was not copied forward in a previous iteration, the act of excluding a wasted copy makes new locations for data available, perturbing the sequence of operations that follow. This makes the record from the previous run only partly useful for avoiding mistakes in the subsequent run, an example of the "butterfly effect."

Still, writing in a container-optimized manner can improve on the naive data placement of M^+, which uses the FTL to fill a container at a time. As an example, consider a sequence in which accesses alternate for a while between data that will be re-read "soon" and "later",

and then never again. Imagine that there is a container's worth of "soon-re-read" data and another container's worth of "later-re-read" data, and there are two free containers. If the two are interspersed as data arrives, each of the two containers in flash will contain 50% "soon" and 50% "later" blocks. Once "soon" arises, and the "soon" blocks are no longer needed, it is necessary to GC and consolidate all the "later" blocks in one container to free up the other container for reuse. On the other hand, by segregating the "soon" and "later" blocks as they are written to flash (E in Table 6.1), we may be able to erase one container, without the need to CF, as soon as all the blocks within it are no longer needed. We refer to E as *container-optimized* since it explicitly consolidates data that die together at the host or application level.

Since the purpose of our study is to provide a best-case comparison point for real-world cache replacement algorithms, we focus henceforth on the container-based cache replacement policies. Note that if containers consist of only one block, any approaches that are specifically geared to containers should work for the unit caching replacement policy. In the next section, we describe the algorithms in greater detail, using **C** to represent the offline heuristic **H** in the context of containers.

6.2.4 Impact from Dirty Data

The results from the various algorithms depend significantly on how the cache treats writes into the file system. For example, Pannier [146] requires that all file writes be inserted into the cache, with the expectation that the cache be an internally consistent representation of the state of the file system at any given time. All writes are immediately reflected in stable storage, which is appropriate for an online algorithm; Pannier's comparison to Belady used the same approach even with future knowledge, writing all user-level writes into SSD.

With future knowledge, however, one can argue that a "dead write" that will be overwritten before being read need not be inserted into the cache. The same is true of a write that is never again referenced, though in that case it should be written through to disk. Since we model only the flash cache RHR and endurance, we place these dead writes into DRAM but not into flash.

6.2.5 Algorithm Granularity

For the remainder of this paper, we compare the following algorithms. **M** refers to variants of MIN while **C** refers to container-optimized algorithms. A table summarizing these (and other) algorithms appears in §6.4.3.

M Belady's MIN, which does not insert a block that will be overwritten or never reread by the client.

M+ A variant of MIN, which identifies a block that is inserted into cache but evicted before

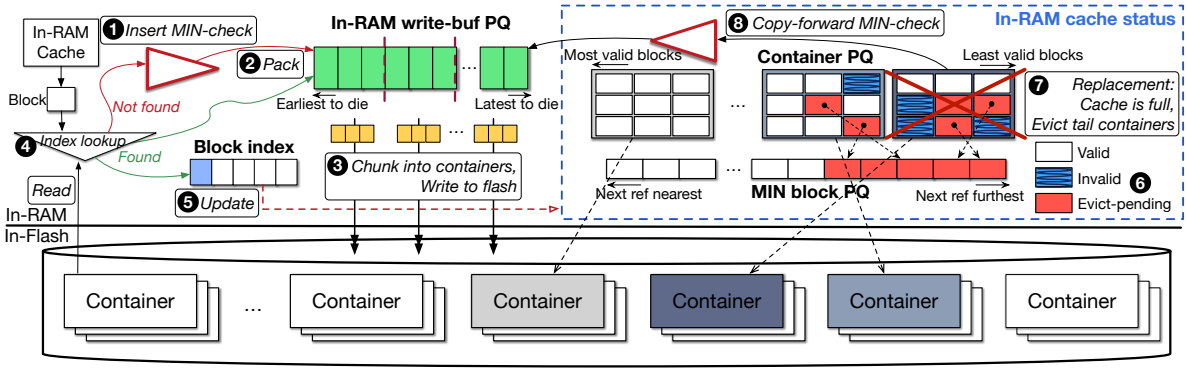


Figure 6.2: Container-optimized offline flash caching framework. PQ: priority queue.

being read, and omits that insertion. Pannier [146] uses something similar to this for its comparison to MIN, by approximating the working set of unique blocks the cache will encounter before reuse. It also uses TRIM to avoid CF once the last access to a block occurs.

$\mathbf{M}_N$ A variant of M^+, which does not insert blocks with accesses $< N$. M_1 is equivalent to M^+, while M_N generalizes it to $N > 1$.

$\mathbf{M}_T$ A variant of M^+, which eliminates enough low-reuse cache insertions to get the best RHR under a specific erasure limit (see §6.2.2).

C Each block is inserted if and only if M^+ would insert it. However, a write buffer of W containers is set aside in memory, and the system GCs whenever it is necessary to make that number of containers free to overwrite. Containers are filled using the *eviction timestamp* indicating when, using M^+, a block would be removed from the cache. The contents of the containers are ordered by eviction time, so the first container has the m blocks that will be evicted first, the next container has the next m blocks, and so on.

$\mathbf{C}_N$, $\mathbf{C}_T$ Analogous to M_N, M_T.

6.3 Offline Approximation

Here we describe how to evolve the container-oblivious MIN algorithm to a container-optimized heuristic C, which provides the same RHR but significantly fewer erasures. We also explain creating C_N and C_T.

6.3.1 Key Components

Figure 6.2 depicts the framework architecture and examples of the insertion and lookup paths. A detailed discussion appears in the next subsection, but the following components are the major building blocks.

Block Index An in-memory index maps from a block's key (e.g., LBA) to a location in flash. Upon a read, the in-memory index is checked for the block's key, and if found, the flash location is returned. Newly inserted blocks are added to the in-memory write buffer queue first. Once the content of the writer buffer is persisted in the flash cache, all blocks are assigned an address (an index entry) used for reference from the index. When invalidating a block, the block's index entry is removed.

In-RAM Write Buffer An in-memory write buffer is used to hold newly inserted blocks and supports in-place updates. The write buffer is implemented as a priority queue where the blocks are ranked based on their eviction timestamps (described in greater detail in §6.3.2). The write buffer queue is filled cumulatively and updated in an incremental fashion. Once the write buffer is full, its blocks are copied into containers, *sealed* and *persisted* in the flash cache. The advantage of cumulative packing and batch flushing is that the blocks with close eviction timestamps get allocated to the same container so that erasures are minimized. Overwrites to existing blocks stored in flash are redirected to the write buffer without updating the sealed container.

In-RAM Cache Status A few major in-memory data structures construct and reflect the runtime cache status. Once a container is sealed, its information structure is inserted into a **container priority queue** (PQ), a structure to support container-level insertion and eviction. Whenever a container is updated (e.g., a block is invalidated or evict-pending, etc.), its relevant position in the queue is updated. In addition to the container PQ, a **block-level MIN PQ** is designed to support the extended Belady logic and track the fine-grained block-level information. We discuss the operations of the MIN PQ in §6.3.2.

6.3.2 Container-optimized Offline Heuristic

The multi-stage heuristic C offers the optimal RHR while attempting to approach the practically lowest number of erasures on the flash cache. In the following, we describe the container-optimized heuristic pipeline (Figure 6.2) in detail. Figure 6.3 shows the pseudocode of how the offline heuristic handles different events.

Insert, Access and Seal We describe inserting a block, accessing a block, and sealing/persisting a container. ❶ When a client inserts a block upon a miss and the cache is not full, the `OnInsert` function first checks if the block's next reference timestamp is `INFINITE` (i.e., the block is never read again in the future or the next reference is a write). If so, `OnInsert` simply bypasses it and returns. Otherwise, an insertion record is checked to see if

```
1 void Lookup(Object obj):
2   // WB: Write buffer priority queue
3   if WB.exist(obj.key) or INDEX.exist(obj.key):
4     Object existing = Read(obj.key)
5     OnAccess(existing, obj)
6   else: OnInsert(obj) // Upon a miss
7
8 void OnInsert(Object obj):
9   if not MINFull():
10    if obj.next_ref == INF: return
11    if Rec[obj.key].read_freq <= read_freq_thresh:
12      return
13    MIN_Q.insert(obj) // Insert into MIN queue
14  else:
15    Object victim = MIN_Q.top()
16    if obj.next_ref > victim.next_ref:  return
17    if Rec[obj.key].read_freq <= read_freq_thresh:
18      return
19    // Trigger evict on MIN queue
20    EvictMIN(MIN_Q, victim)
21    MIN_Q.insert(obj)
22  if WB.full(): OnSeal()
23  WB.insert(obj) // Pack into WB
24  EvictFlash()
25
26 void OnSeal():
27   Object obj = WB.begin()
28   // Iterate through all sorted objs
29   while obj != WB.end():
30     FreeCList[curr_ptr].insert(obj)
31     if FreeCList[curr_ptr].full():
32       // C_Q: Container queue
33       C_Q.insert(FreeCList[curr_ptr++])
34     obj = WB.next()
35   WB.clear()
36
37 void EvictMIN(Object victim):
38   MIN_Q.pop();
39   if WB.exist(victim.key): // Remove if in WB
40     WB.erase(victim)
41     return
42   Container c = GetContainer(victim)
43   victim.evict_pending = true
44   c.num_evict_pending++
45   C_Q.update(c) // Update c's position in C_Q
46
47 void EvictFlash():
48   while FlashFull():
49     Container c = C_Q.pop()
50     GC(c) // Garbage collect the evicted c
51
52 void OnCopyForward(Object obj):
53   if obj.next_ref == INF: return
54   MIN_Q.insert(obj)
55
56 void OnAccess(Object old_obj, Object new_obj):
57   if old_obj.evict_pending:
58     Count access as a miss
59     return
60   Update hit stats
61   old_obj.next_ref = new_obj.next_ref
62   // Update obj's position in MIN queue
63   MIN_Q.update(old_obj)
64   old_obj.evict_time = new_obj.evict_time
```

Figure 6.3: Functions handling events for flash-cached blocks and containers in the container-optimized offline heuristic.

the block exceeds `read_freq_thresh`, a configurable read hit threshold. For instance, setting the `read_freq_thresh` to 1 filters out those that are inserted but evicted before being read, avoiding a number of *wasted* writes. C_N and C_T also take advantage of this scheme for trading-off RHR with endurance: setting the threshold higher additionally filters out the less *useful* writes, reducing RHR but decreasing the number of erasures. The threshold can be fixed (C_N) or computed based on the total erasure budget (C_T, as described for H_T in §6.2.2). More useful writes, which result in a greater number of read hits, still take place. We study the trade-offs in §6.5.3.

Once the threshold test succeeds, if the cache is full and the block will be referenced furthest in the future (i.e., the block has a greater next reference timestamp than the most distant block (`victim`) currently stored in the cache), `OnInsert` returns without inserting it. When both checks are passed, `EvictMIN` function is triggered to evict the `victim` and the new block

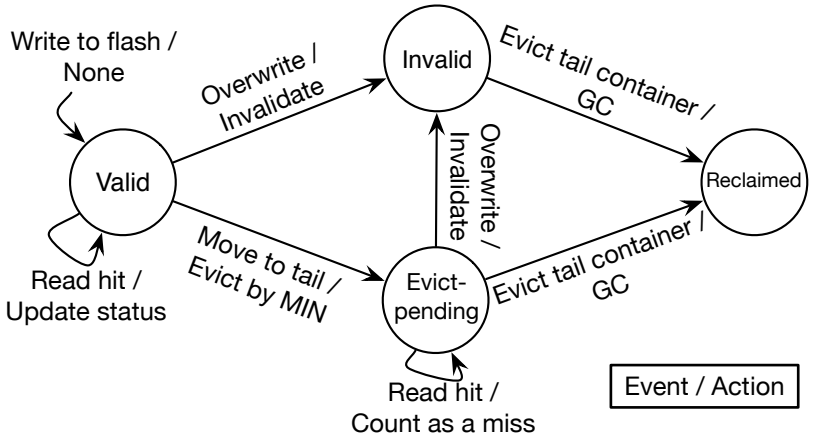


Figure 6.4: State transitions of blocks in our heuristic.

is inserted into the MIN queue (`MIN_Q`). At the same time, ❷ the block is added to the in-memory write buffer queue (`WB`).

❸ When `WB` is full, all the blocks held in it, sorted based on their eviction timestamp, are copied into multiple containers from the free container list `FreeCList`. `curr_ptr` maintains a pointer to the first available free container in `FreeCList`. (We compare the sorted approach to FIFO insertion of the blocks in evaluation as well.) The `OnSeal` function then persists the open containers in the flash cache.

Lookup ❹ On a `Lookup`, both the `WB` and in-memory index (`INDEX`) are referenced to locate the block, and the read is serviced. On a read access (`OnAccess`), ❺ the read hit updates the existing block's block-level metadata (`old_obj`) and `old_obj`'s position is updated in `MIN_Q` on the next access time. Upon a miss `OnInsert` is triggered as described.

Invalidation and Eviction ❻ The container-optimized offline caching introduces another new block state – *evict-pending*. Evict-pending describes the state when a block is evicted from `MIN_Q` (transitioning from the *valid* state to evict-pending) but temporarily resides in the GC area of the flash, pending being reclaimed. Figure 6.4 shows the state transitions of a block in the heuristic. A block is inserted/reinserted into the flash cache with a valid state. Once it is overwritten, the old block in the flash is marked as invalid and the updated data is inserted into `WB`. Overwriting an evict-pending block makes it transition to the invalid state. If the `victim` to be evicted from `MIN_Q` happens to reside in `WB`, `EvictMIN` directly removes it from the memory. The on-flash container maintains a `num_evict_pending` counter. On evict-pending, the corresponding container increments its counter and updates its position in the container PQ `C_Q`.

Let V, I, and E represent the percentage of valid, invalidated and evict-pending blocks in a container, respectively; then $V + I + E = 100\%$. The priority of a container is calculated

using V. When the cache is full, `EvictFlash` selects the container with the lowest V (i.e., the fewest valid blocks) for eviction.

Copy-forwarding and GC ❼ When the cache is full and a container has been selected for eviction, the heuristic copies valid blocks forward to the in-memory write buffer. Function `OnCopyForward` is called to check if the reinserted block is useful. All the invalidated and evict-pending blocks get erased in the flash. The selected container is then reclaimed and inserted back to `FreeCList`.

❽ The check for a "useful" reinserted block looks for future references and (optionally) confirms the block will not be evicted before it is read.

6.4 Experimental Methodology

Throughout our analyses, we set the flash erasure unit size to 2MB. We place a small DRAM cache (5% of the flash cache size) in front of the flash cache to represent the use case where the flash cache is used as a second-level cache. (The DRAM cache uses the MIN eviction policy for offline flash cache algorithms and LRU for online ones.) Because some of the datasets in the repositories we accessed have too small a working set for 5% of the smallest flash cache to hold the maximum number of in-memory containers, we restrict our analyses to those datasets with a minimum 32GB working set.

This section describes the traces; implementation and configuration for the experimental system; and the set of caching algorithms evaluated.

6.4.1 Trace Description

We use a set of 34 traces from 3 repositories:

EMC-VMAX Traces: This includes 25 traces of EMC VMAX primary storage servers [184] that span at least 24 hours, have at least 1GB of both reads and writes, and meet the minimum working set threshold (slightly over half the available traces).

MS Production Server Traces: This includes 3 storage traces from a diverse set of Microsoft Corporation production servers captured using event tracing for windows instrumentation [126], meeting the 32GB minimum.[2]

MSR-Cambridge Traces: This includes 6 block-level traces lasting for 168 hours on 13 servers, representing a typical enterprise datacenter [170]. We narrowed available traces to 6 to include appropriate traces for cache studies. The properties include a working set size greater than 32GB, $\geq 5\%$ of capacity accessed, and read/write balance ($\leq 45\%$ writes).[3]

[2] The traces are: `BuildServer`, `DisplayAdsPayload`, `DevelopmentToolsRelease`.

[3] The traces are: `prn0`, `prn1`, `proj0`, `proj4`, `src12`, `usr2`.

Given a raw trace, we annotate it by making a full pass over the trace and marking each 4KB I/O block with its next reference timestamp. Large requests in traces are split into 4KB blocks, each of which is annotated with the timestamp of its next occurrence individually. The annotated trace is then fed to the cache simulator. Round 1 simulation, e.g., M, may generate the insert log that can be used by round 2 simulation (e.g., M^+, M_N) to filter out blocks that will be evicted before being read.

6.4.2 Implementation and Configuration

For the experimental evaluation, we built our container-optimized offline caching heuristic by adding about 3,900 lines of C++ code to a full-system cache simulator. It reports metrics such as hit ratio and flash erasures based on the Micron MLC flash specification [168].

The size of the flash cache for each trace is determined by a fixed fraction of the WSS of the trace (from 1–10%). For C, a write buffer queue (with default size equal to 4 containers) is used for newly inserted blocks and is a subset of this DRAM cache. We over-provision the flash capacity by 7% by default; this extra capacity is used for FTL GC or to ensure the ability to manually clean containers in the container-optimized case. We discuss varying the over-provisioning space in §6.5.2.

We conduct the simulation study on 4 VMs each equipped with 4 cores and 128 GB DRAM. All tests are run in parallel (using `GNU parallel` [94]). We measured the CPU time of heuristic M^+ and C looping over all traces, not including the runtime of trace annotating and insertion log generating pre-runs. We ran the experiments 5 times and variance was low. C takes 21.7% longer (2.47 hr) than M^+ (2.03 hr) for the smallest cache size due to the overhead of PQ used by C under intensive GCs. The heuristic keeps track of more metadata in 10% cache size. Thus, with the least amount of GCs, it takes M^+ almost as long (2.19 hrs) as C does (2.21 hrs), to replay all traces. The results show that our heuristic simulation can process a large set of real-world traces within a reasonable time. This strengthens our confidence that our offline heuristics can serve as a practically useful tool.

6.4.3 Caching Algorithms

Table 6.2 shows the caching algorithms selected to represent past and present work. We select the classic LRU algorithm, two state-of-the-art container-based online algorithms (described next), and a variety of offline algorithms (as described in §6.3). The configurations for previous work are the default in their papers (e.g., number of queues) unless otherwise stated.

RIPQ+ is based on RIPQ [188], a container-based flash caching framework to approximate several caching algorithms that use queue structures. As a block in a container is accessed, its ID is copied into a virtual container in RAM, and when a container is selected for eviction, any blocks referenced by virtual containers are copied forward. We adopt a modified version

Policy	Abbrev.	Description	O	C
LRU	L	least recently used	✗	✗
RIPQ+	R^+	RIPQ with overwrites, segmented-LRU	✗	✗
Pannier	P	container-based, S2LRU*, survival queue, insertion throttling	✗	✓
MIN	M	FK, do not insert data whose next ref. is the most distant	✓	✗
MIN+	M^+	FK, do not insert data evicted without read (R_1+TRIM, **O1**)	✓	✓
MIN+write-threshold	M_T	FK, limit number of insertions (R_T+TRIM, **O2**)	✓	✓
MIN+insertion-removal	M_N	FK, do not insert data with accesses $< N$ (R_N+TRIM, **O3**)	✓	✓
Container-optimized	C	FK, container-optimized (R_1+TRIM+CFR+E, **O1**)	✓	✓
C+write-threshold	C_T	FK, container-optimized (R_T+TRIM+CFR+E, **O2**)	✓	✓
C+insertion-removal	C_N	FK, container-optimized, do not insert data w/ acc. $< N$ (R_N+TRIM+CFR+E, **O3**)	✓	✓

Table 6.2: Caching algorithms. FK: future knowledge, O: offline, C: container-optimized.

of RIPQ that handles overwrite operations, referred to as RIPQ+ [146]. In our experiments, we use the segmented-LRU algorithm [123] and a container size of 2MB with 8 insertion points.

Pannier [146] is a container-based flash caching mechanism that identifies divergent (heterogeneous) containers where blocks held therein have highly varying access patterns. Pannier uses a priority-based survival queue to rank containers based on their survival time, and it selects a container for eviction that has either reached the end of its survival time or is the least-recently used in a segmented-LRU structure. During eviction, frequently accessed blocks are copied forward into new containers. Pannier also uses a multi-step credit-based throttling scheme to ensure flash lifespan.

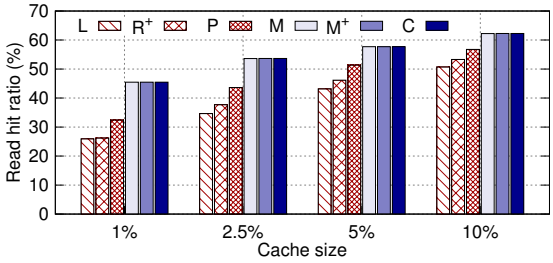
6.5 Evaluation

In §6.5.1 we evaluate a number of caching algorithms with respect to RHR and EPBPD. We also evaluate the contribution of various techniques on the improvement in endurance. This is followed in §6.5.2 by a sensitivity analysis of some of the parameters, the use of the write buffer and overprovisioning. We then consider tradeoffs that improve endurance at a cost in RHR (§6.5.3).

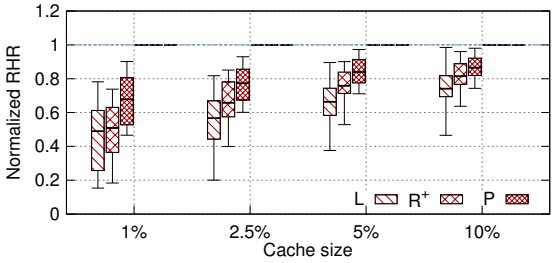
6.5.1 Comparing Caching Algorithms

We first compare the three online algorithms from Table 6.2 with the RHR-maximizing offline algorithms: MIN, M^+ and C. We focus initially on **O1**, minimizing erasures subject to a maximal read hit ratio. Figure 7.5 shows the RHR and EPBPD results across all 34 traces, while varying the cache size among 1%, 2.5%, 5% and 10% of the working set size for each trace.

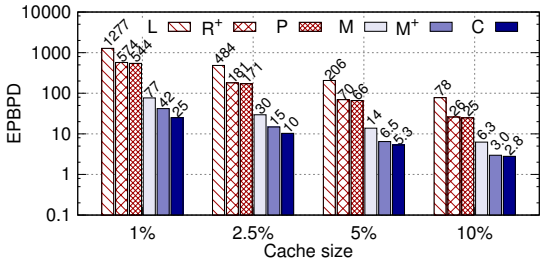
In §7.4.3, we see that LRU (the left bar in each set) obtains the lowest hit rate because it has neither future knowledge nor a particularly sophisticated cache replacement algorithm. RIPQ+ has about the same RHR as LRU; its benefits arise in reducing erasures rather than



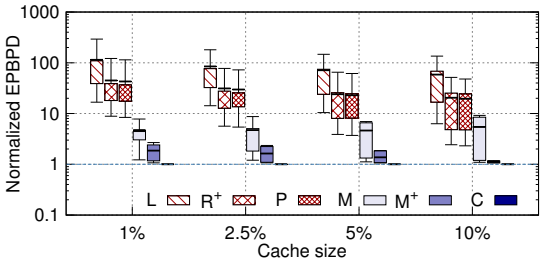
(a) Read hit ratio.



(b) Normalized read hit ratio. $M/M^+/C$ are all 1 due to normalization.



(c) EPBPD.



(d) Normalized EPBPD.

Figure 6.5: RHR and EPBPD for various online/offline caching algorithms and sizes. EPBPD is shown on a log scale, with values above the bars to ease comparisons. The box-and-whisker plots in (b) and (d) show the {10, 25, average, 75, 90}%-ile breakdown of the normalized RHR and EPBPD, respectively; each is normalized to that of the best-case C.

increasing hit rate. Pannier achieves up to 26% improvement in RHR over LRU and RIPQ+. By leveraging future knowledge, MIN, M^+ and C achieve the (identical) highest hit ratio, which improves upon Pannier by 9.7%–40%. The gap is widest for the smallest cache sizes. §7.4.3 shows the range of the normalized RHR (normalized against the best-case C), with a similar trend as shown in §7.4.3.

Figure 6.5(c) and 6.5(d) show that online algorithms incur the most erasures. Pannier performs slightly better than RIPQ+ due to the divergent container management and more efficient flash space utilization. Though it is not explicitly erasure-aware, MIN saves up to 86% of erasures compared to Pannier. This is because with perfect future knowledge, MIN can decide not to insert blocks that would never be referenced or whose next reference is a write. This implicit admission control mechanism results in significantly fewer flash erasures and higher RHR. M^+ further reduces erasures by 31% compared to MIN, because M^+ avoids inserting blocks that would be evicted before being accessed and uses TRIM to avoid copying blocks during GC if they will not be rereferenced. Variation does exist, as shown in Figure 6.5(d): the 10% and 25%-ile (the lower bound of box and whiskers) breakdown of M^+ are closer to 1 than those of MIN; the lower bounds never reach below

1 while the upper bounds (75% and 90%-ile) are far above, especially for small cache sizes. At small cache sizes, the container-optimized offline heuristic, C, further improves on M^+ by 40% by lowering GC costs.

One way to understand the differences among MIN, M^+, and C is to view the contributions of the individual improvements. The cumulative fraction of erasures saved by the techniques used by M^+ or C (summarized in Table 6.1) is depicted in Figure 6.6, normalized against the erasures used by MIN. Surprisingly, we find that simple techniques such as R_1 and TRIM have a greater impact on reducing erasures compared to more advanced techniques such as the container-optimized E.

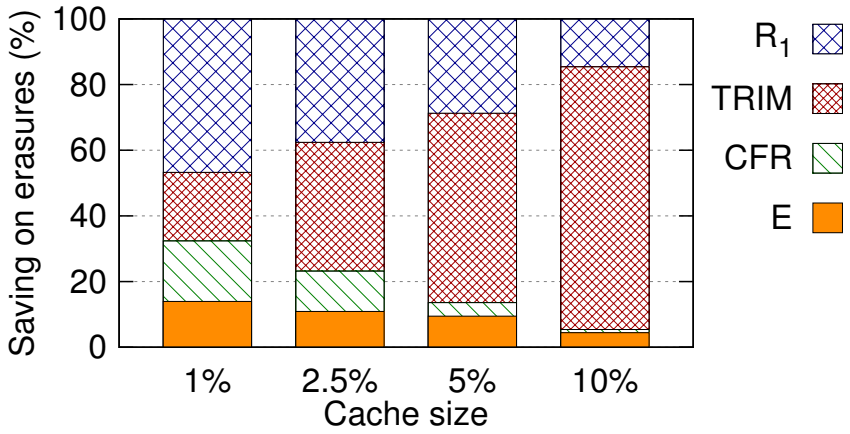


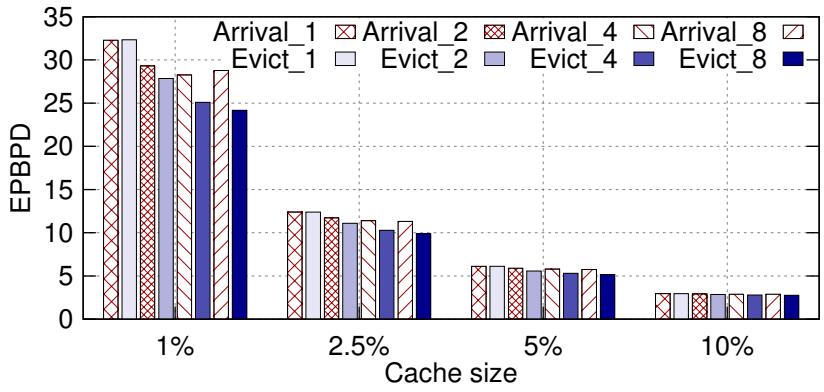
Figure 6.6: Fraction of erasures saved in different stages. R_1: never inserting blocks that will be evicted before being read. TRIM: removing blocks that would not be reread at FTL layer, CFR: avoiding wasted CFs, E: packing blocks in write buffer using eviction timestamp.

- The top component in each stacked bar (**R_1**) shows the relative improvement from preventing the insertion of blocks that will be evicted without being referenced; for the smallest cache, this accounts for about half the overall improvement from all techniques, but it is a relatively small improvement for the largest cache.
- For MIN, using **TRIM** to avoid the FTL copying of blocks that will not be reaccessed has an enormous contribution for the largest cache (˜80% of all erasures eliminated), but it is a much smaller component of the savings from the smallest caches. Note that we convert MIN to M^+ by avoiding (1) unread blocks due to eviction and (2) FTL GC for unneeded blocks.
- Adding a check for blocks that are copied forward (**CFR**) but then evicted without being rereferenced has a moderate (10%) impact on the smallest cache, but little impact on the largest. This is done only for C, as the copy-forwarding within the FTL for M^+ occurs in a separate component.

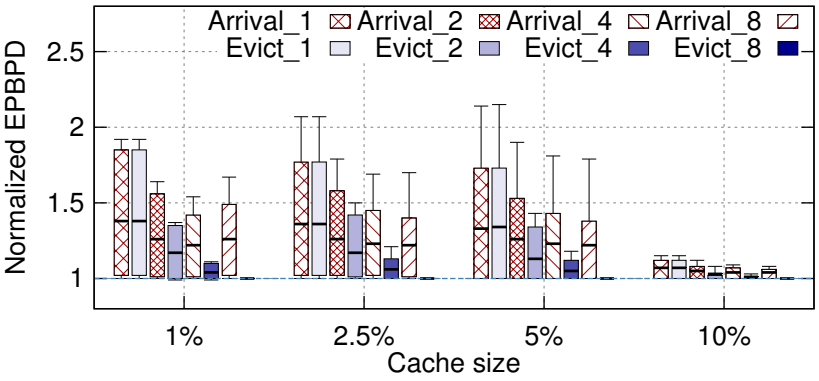
- Using 4 write buffers and grouping by eviction timestamp (**E**) has a similar effect to **CFR** on smaller caches and a nontrivial improvement for larger ones.

6.5.2 Sensitivity Analysis

Thus far we have focused on default configurations. Here we compare the impact of different in-memory write buffer designs and sizes on erasures. Then we examine the impact of over-provisioned capacity.



(a) EPBPD.



(b) Normalized EPBPD.

Figure 6.7: Impact of consolidating blocks based on eviction timestamp, and trade-offs in varying the size of the in-memory write buffer (a multiple of containers); `Arrival_2` means packing the blocks into the write buffer (2-container worth of capacity) based on their arrival time, `Evict_4` means packing the blocks into the 4-container-sized write buffer based on the eviction timestamp. The box-and-whisker plot in (b) shows the {10, 25, average, 75, 90}%-ile breakdown of the EPBPD, normalized to that of `Evict_8`.

Impact of Consolidating Blocks

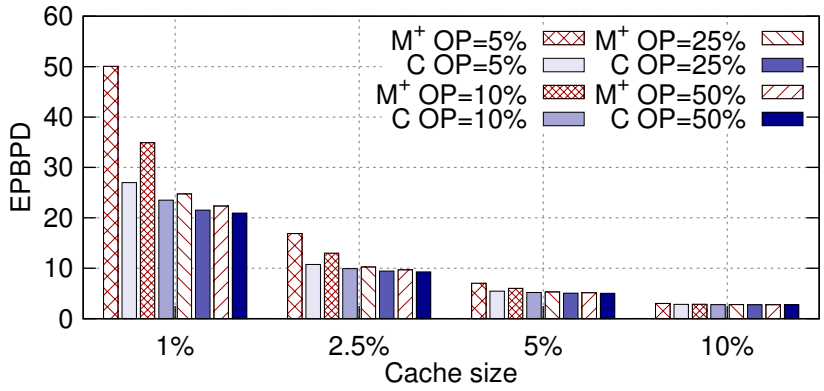
We study the impact of consolidating blocks with similar eviction timestamps into the same containers and the impact of sizing the write buffer. Figure 6.7 plots the average EPBPD and variation across all traces, as a function of policy and write buffer size, grouped by different cache sizes. All experiments are performed using C and give the same optimal RHR. By default C uses the priority-based queue structure as the in-memory write buffer, which is filled using the eviction timestamp indicating when, using M^+, the block would be evicted from the cache. The write buffer, once full, is dispersed into containers that are written into flash. For comparison purposes we implemented a FIFO-queue-based write buffer, where the blocks are simply sorted based on their arrival timestamp.[4] There is no difference in EPBPD with a queue size of 1 container, because blocks cannot be sorted by eviction timestamp with a single open container in DRAM. Increasing the write buffer size, we observe a reduction in erasures with `Evict`. This effect is more pronounced with a bigger write buffer queue. For example, for the 2.5% cache size, `Evict_2` reduces the EPBPD by 5% compared to `Arrival_2`; but this EPBPD differential increases to 13% for an 8-container write buffer. This is because a bigger `Evict` write buffer results in less fragmentation due to evict-pending blocks in containers stored in flash. The trend shown in average consistently matches that of the individual variation in Figure 6.7(b). Interestingly, `Arrival_8` with a 1% cache size yields a slightly higher EPBPD than that of `Arrival_4`. This is because the fraction of data copied forward internally (due to GC) is higher when using a relatively small cache and a large write buffer. We observed that while the 4 least populated containers generally were sufficiently "dead" to benefit from GC, the next 4 (5^{th}–8^{th} least populated) containers hold significantly more live data than the first four when collected in a batch; this increases CF significantly. This further demonstrates that consolidating blocks into containers based on their eviction timestamp can effectively reduce erasures.

Impact of Over-provisioned Capacity

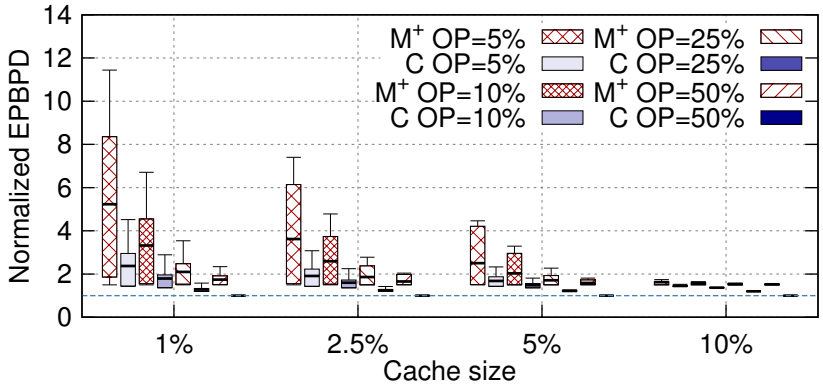
Next, we analyze the impact of over-provisioned (OP) capacity on erasures. Figure 6.8 shows the average EPBPD when varying the over-provisioned space between 5% and 50% for different cache sizes. As in the previous experiment, we omit results of RHR, which are unaffected by overprovisioning. We classify the results into block-level (M^+) and container-optimized (C) approaches, which are interspersed and grouped by the amount of over-provisioned capacity. (Thus, for a given capacity, it is easy to see the impact of the container-optimized approach.)

For both groups, a larger OP space results in lower EPBPD, because the need for GC to reclaim new space becomes less urgent than a flash equipped with relatively smaller OP space. This effect is more significant for M^+, as M^+ manages data placement in a container-oblivious manner; this results in more GCs, which in turn cause larger write amplification (more internal writes at the FTL level). Comparing M^+ with C, we observe that for a 1% cache size, C with 10% OP incurs fewer erasures than M^+ with 25% OP. This is because

[4]Packing blocks based on LBA or whether clean (newly inserted) or dirty (invalidated due to overwrite), yields no impact on erasures or WFUE scores for the offline heuristics.



(a) EPBPD.



(b) Normalized EPBPD.

Figure 6.8: Trade-offs in over-provisioned (OP) space. The box-and-whisker plot shows normalized EPBPD against `C OP=50%`.

C consolidates blocks that would be removed at roughly the same time, resulting in significantly fewer internal (CF) flash writes. *C* also avoids CF of most blocks that get evicted before reaccess. With the largest cache, however, the relative benefit from additional over-provisioning is nominal. Again, Figure 6.8(b) demonstrates that the variation across traces exists; and it is just a matter of how many more erasures each trace incurs, compared to the best case.

6.5.3 Throttling Flash Writes

Thus far the evaluation has focused on **O1**, optimizing first for RHR and second for EPBPD. If erasures are more important, we can limit flash writes to reduce EPBPD at some cost to hit rate. Recall that **O2** tries to maximize RHR subject to a specific limit, whereas **O3** tries to optimize WFUE given a particular weight of the importance of RHR relative to EPBPD.

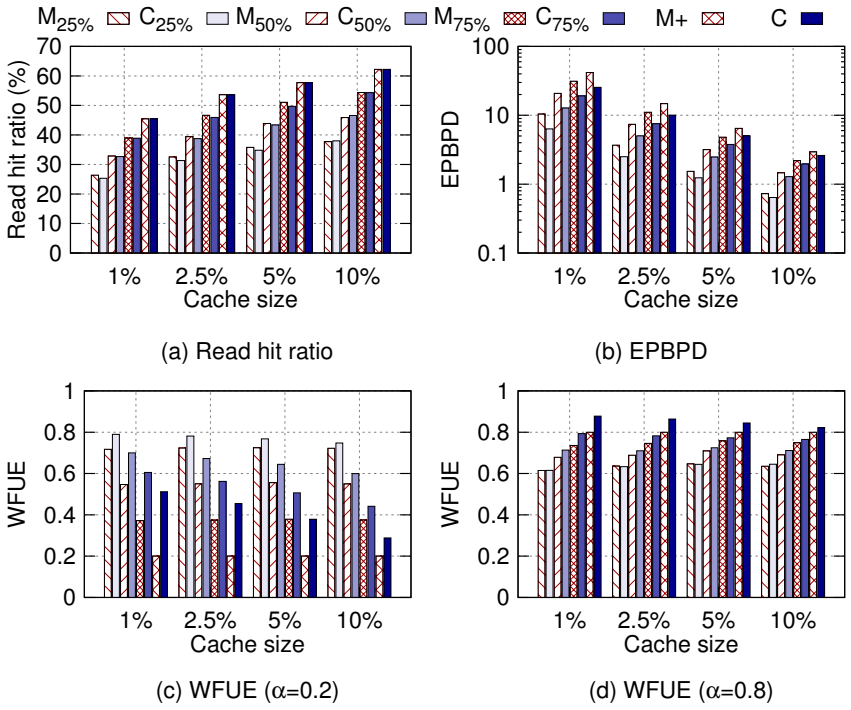


Figure 6.9: Trade-offs when limiting erasures: WFUE as a function of algorithm, EPBPD quota, cache size, and α weights.

Figure 6.9 demonstrates the effect of C_T, which uses the admission control logic described in §6.2.5 to meet a specific EPBPD limit. It removes insertions with the least impact (i.e., blocks with least number of read hits) on RHR to meet the endurance goal. Figure 6.9 shows the results averaged across all traces when varying the EPBPD quota for a trace from 25%–75% of the EPBPD necessary for M or C respectively; this represents a reasonable range of user requirements on flash lifespan in real-world cases. For each cache size there are eight bars, with pairs of M_T and C_T algorithms as the threshold varies from 25% to 100% of total erasures. (The limits for M_T and C_T are set differently, since the maximum values vary.)

We observe in Figure 6.9(a) that for big cache sizes (5% and 10%) the RHR loss is about 39% for the 25% EPBPD quota. The gap reduces to 13% for the 75% EPBPD quota. As expected and shown in Figure 6.9(b), overall EPBPD decreases as the threshold is lowered, while C_T moderately improves upon M_T.

For WFUE, one question is what an appropriate weight α would be. Figures 6.9(c) and (d) plot the same algorithms but report WFUE using $\alpha = 0.2$ and $\alpha = 0.8$ (this prioritizes erasures and hit rates respectively). With $\alpha = 0.2$, the erasure savings dominate. Hence, $M_{25\%}$ and $C_{25\%}$ achieve the highest WFUE scores while M^+ and C see lower ones. Prioritizing

RHR gives M^+ and C the highest WFUE across all variants. C_T consistently outperforms the corresponding M_T because it can avoid most wasted CFs and because it groups by eviction timestamp (see §6.5.1).

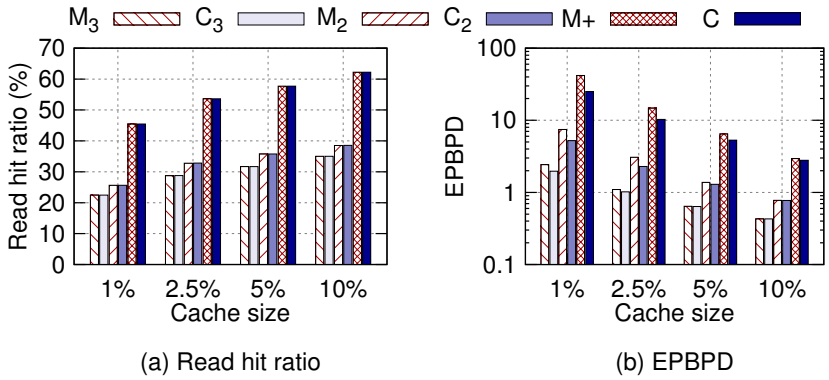


Figure 6.10: Trade-offs in read hit based insertion removals.

Figure 6.10 shows the effect of limiting flash insertions to blocks that are reread $\geq N$ times. It plots M^+ (which is equivalent to M_1), M_2, and M_3, as well as the corresponding container-optimized C algorithms. Results for RHR and EPBPD are averaged across all 34 traces. For small cache sizes (1% and 2.5%), C_2 loses an average of 41% of the RHR (Figure 6.10(a)), but it gets about a 79% savings in EPBPD (Figure 6.10(b)). Prioritizing erasures and RHR shows similar trends as the WFUE results in read hit based insertion removal tests (Figures 6.9(c) and (d)), hence are omitted due to space constraints.

6.6 Chapter Summary

While it is challenging to optimize for both RHR and endurance (represented by EPBPD) simultaneously, we have presented a set of techniques to improve endurance while keeping the best possible RHR, or conversely, trade off RHR to limit the impact on endurance. In particular, our container-optimized heuristic can maintain the maximal RHR while reducing flash writes caused by garbage collection; we see improvements of 55%–67% over MIN and 6%–40% over the improved M^+, which avoids many wasted writes and uses TRIM to reduce GC overheads. Another important finding in our study indicates that simple techniques such as R_1 and TRIM provide most of the benefit in minimizing erasures. Alternatively, the flash writes can be limited to those that are rereferenced a minimum number of times. We define a new metric, Weighted Flash Usage Effectiveness, which uses the offline best case as a baseline to evaluate tradeoffs between RHR and EPBPD quantitatively.

In the future, we would like to investigate the complexity of the various algorithms (we believe them to be NP-hard). Exploring approaches to improving online flash caching algorithms

is also part of our future work. We are particularly interested in heuristics to trade off one cache hit against another to further reduce cache writes without impacting RHR.

Chapter 7

A Framework for Building Distributed Key-Value Stores

The workload-aware key-value storage systems presented in Chapter 3 and 5 require a significant amount of engineering effort to implement, test, and deploy. To address this problem, in this chapter we present a flexible and generic framework for fast prototyping a wide range of distributed key-value store designs. The resulted framework, ClusterOn, targets a proxy-based layered distributed key-value storage architecture, and adds programmability to the control plane, which enables many different practical key-value store design and configurations.

7.1 Introduction

The big data boom is driving the development of innovative distributed storage systems aimed at meeting the increasing need for storing vast volumes of data. We examined the number of representative storage systems that have been implemented/released by academia in the last decade, and found a steady increase in such systems over recent years. Figure 7.1 highlights the trend of innovating new solutions for various and changing storage needs.[1]

These new storage systems/applications[2] share a set of features such as replication, fault tolerance, synchronization, coordination, and consistency. This implies that a great portion of these features are overlapped across various such applications. Furthermore, implementing a new application from scratch imposes non-trivial engineering efforts in terms of # lines of code (LoC) or person-year. Table 7.2 gives the LoC[3] of 6 popular distributed storage

[1] We only count papers that implement a full-fledged storage system. USENIX FAST is not included as it solely covers storage.

[2] We use "systems/applications" interchangeably throughout.

[3] We count the LoC on a per-file basis, excluding source code of the client side and the testing framework.

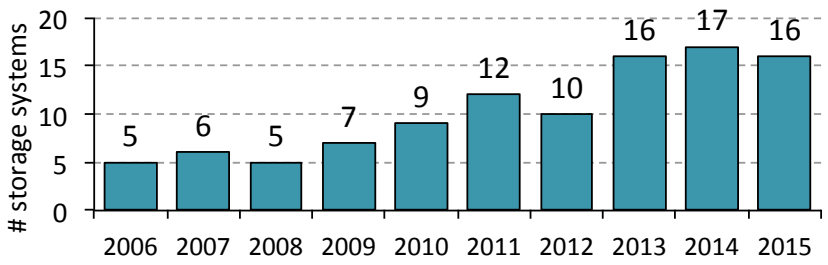


Figure 7.1: Number of storage systems papers in SOSP/OSDI, ATC and EuroSys conferences in the last decade (2006–2015).

	Key-value store			Object store		DFS
	Redis	HyperDex	Berkeley DB	Swift	Ceph	HDFS
Total #LoC	41760	20691	159547	43375	187144	112510

Table 7.1: Total LoC in the 6 studied storage applications.

applications in three categories. Correspondingly, Figure 7.2 shows the LoC breakdown. An interesting observation is that all of the six applications have a non-trivial portion of LoC (45.3%–82.2%) that implements common functionalities such as distributed management. While LoC is a major indicator for the engineering effort involved, it is by no means definitive or comprehensive enough. The engineering effort required at different stages of development also includes the fundamental difficulties of bundling management components as well as increased maintenance cost (e.g., bug fixing) as the codebase size increases. It would be highly desirable and efficient if the common feature implementations across various storage applications can be *"reused"*.

One may argue that different storage solutions are developed to meet certain needs, and thus are specialized. In this paper, however, we posit that storage management software, not the application developers, should implement all the *messy plumbings* of distributed storage applications. We argue that there is a strong need for such a modularized framework that provides a thin layer to realize the common functionalities seen in distributed storage applications. On one hand, such a framework will significantly reduce the complexities of developing a new storage application. On the other hand, a modularized framework enables more effective and simpler service differentiation management on the service provider side.

We envision a framework where *developers only need to implement the needed core functionality*, and common features/management etc. is automatically provided, akin to User Defined Functions in the popular MapReduce framework. To this end, we propose ClusterOn, a framework that takes a non-distributed core version (which we call *datalet*) of a storage application, adds common features and management, and finally morphs the code into a scalable distributed application. ClusterOn is motivated by the observations we made ear-

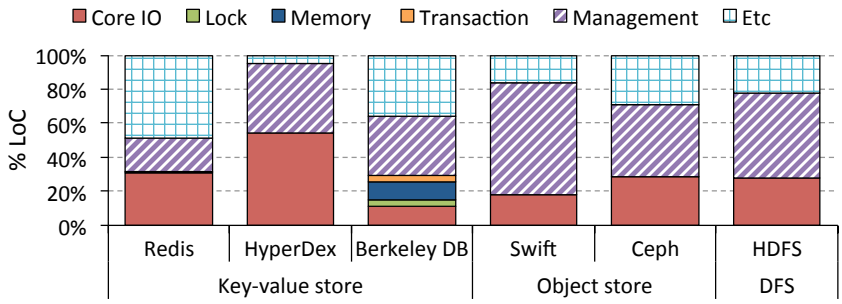


Figure 7.2: LoC breakdown of the 6 studied storage applications. `Core IO` component includes the core data structure and protocol implementation. `Management` component includes implementations of replication/recovery/failover, consistency models, distributed coordination and metadata service. `Etc` includes functions providing configurations, authentications, statistics monitoring, OS compatibility control, etc. `Management` and `Etc` are the components that can be generalized and implemented in ClusterOn.

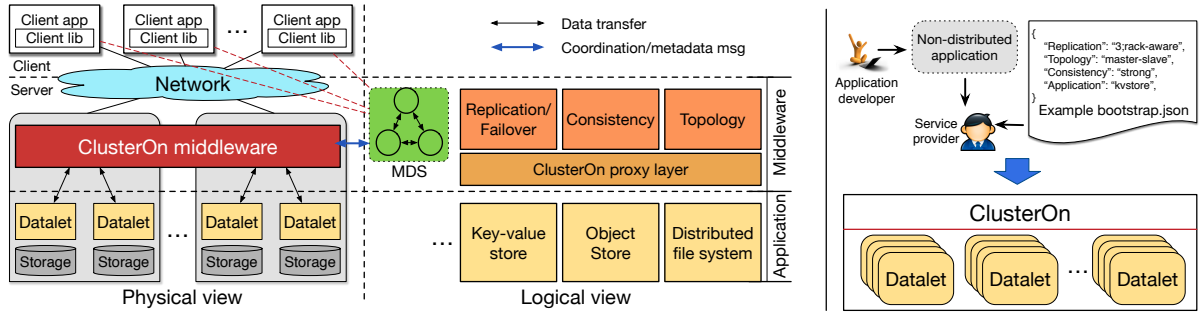
lier that modern storage applications share a great portion of functionality. By providing a transparent and modularized distributed framework that provides configurable services such as replication, fault tolerance, and consistency, ClusterOn hides the inherent complexities of developing distributed storage applications.

This paper makes the following contributions. We quantitatively analyze various distributed storage application and show that they have repetitive source code that implements common features found across these applications. As a novel solution, we present the design of ClusterOn, which provides these features to reduce the engineering effort required for development of new distributed applications.

7.2 ClusterOn Design

Figure 7.3(a) shows the architecture of ClusterOn. It consists of three major components—(1) application layer, (2) middleware, and (3) a metadata server.

The ***application layer*** includes a homogeneous cluster of *datalets*. A datalet is a single instance of the application. These datalets are the building blocks of constructing larger distributed applications, although they are completely unaware that they are running in a distributed setting. Datalets are designed to run on a single node, and they are only responsible for performing the core functions of an application. For example, a KV store datalet, in the simplest form, only needs to implement a `Get` and a `Put` interface. When running a single datalet is no longer sufficient to handle the load, we can instantiate more of them. However, for them to work coherently with one another, the middleware layer is



(a) ClusterOn architecture.

(b) ClusterOn deployment process.

Figure 7.3: ClusterOn architecture.

needed to perform the required coordination.

The ***middleware layer*** has 2 main responsibilities: (1) manages cluster topology, and (2) depending on which, coordinates network traffic amongst the datalets. For scalability reasons, the middleware layer needs to have a distributed set of entities (we call them *proxies*) to perform these 2 functions well on behave of a distributed set of datalets. In the simplest design, one could have a one-to-one mapping between a proxy and a datalet that are symbiotically co-located with all traffic going into and out of the datalet proxies. Proxies will communicate with one another depending on the relationship between the datalets that they proxy within the cluster topology. For example, in a master-slave topology, the master's proxy will forward all write requests to the slave's so the data can be replicated appropriately. In most cases, a single proxy can handle N instances of datalets (where $N \geq 1$, and N is configurable depending on the processing capacity of both the proxy and datalet).

Different applications use different protocols for their communication. ClusterOn's proxies parse the application messages to make decisions such as routing. To support new applications, there are two basic options: (1) the HTTP-based REST protocol; and (2) Google's protocol buffer. Due to the cost that these two solutions may incur, performance-sensitive applications can select to use a simple yet generic text/binary-based protocol, which is also supported by ClusterOn.

The ***meta-data server*** is used by the middleware layer as a persistent store for keeping critical meta-data about the cluster, e.g., topology information, current load on datalets, so the network traffic can be appropriately routed, data can be recovered from a failure, or datalets autoscaled when appropriate. Clients to the cluster can also consult the meta-data server for directory services. This allows clients to more efficiently direct their request to nodes within the cluster across a wide variety of topologies. However, this is optional, as proxies can always redirect requests within themselves to route to the destination datalet or datalets.

7.2.1 Modules

A major challenge in designing ClusterOn is to make sure it can cover various types of distributed applications, e.g., KV stores, object stores, distributed file systems, databases, etc. Though these applications are distributed and share similar modules, they are very different in nature. Hence, there is a need for a systematic approach to classify these applications into different categories and make sure that ClusterOn supports all those categories. ClusterOn realizes this by classifying and supporting these applications on the basis of their underlying replication schemes, cluster topology, and consistency model.

Replication schemes Different applications adopt different replication schemes, e.g., SPORE [102] replicates data at granularity of KV pairs and Redis at node level. File systems such as GFS [91] and HDFS replicate block-level data chunks within and across racks to provide better fault tolerance. ClusterOn's replication module supports a 2-dimensional configuration space: (1) replication granularity, e.g., key/shard/node-level, and (2) replication locality, e.g., server/rack/datacenter-local. ClusterOn provides a generic module that allows flexible selection of different replication schemes, thus covering a wide variety of storage application use cases.

Cluster topologies Cluster topology module specifies the logical relationship among holders (datalets) or different replicas. Generally, most distributed storage applications can be divided in three types of cluster topologies, namely, (1) master-slave, (2) multi-master (or active-active), (3) peer-to-peer (P2P), or some combinations of these. ClusterOn supports the above three topologies. Master-slave mode provides chain replication support [191] by guaranteeing that only one datalet is acting as the master and all others as slaves while keeping this fact oblivious to these datalets. Similarly, in the case of active-active topology, multiple datalets can be concurrently write to and read from, and this can be coordinated by the corresponding proxies using distributed locks. In case of P2P topology, ClusterOn controls the management logic that drives the inter-node communication among the peers. As consistent hashing is commonly used in a P2P topology, ClusterOn can easily calculate who are the immediate neighbors of a datalet for data propagation and recovery purposes.

Consistency models ClusterOn supports three consistency levels that are commonly adopted by the modern distributed storage applications, namely, (1) strong consistency, (2) eventual consistency, and (3) no consistency. In our design, ClusterOn leverages Zookeeper as a distributed locking service to support strong and eventual consistency. Lock can be acquired at granularity of KV pair, block, object, file, or node level. In the case of strong consistency, each incoming request to access data is executed after acquiring a lock on the data. Eventual consistency is supported by first acquiring lock only on the primary copy of the data and updating the secondary copies later.

7.3 ClusterOn Implementation

In this dissertation, we specifically focus on supporting distributed key-value stores within ClusterOn. We have implemented a prototype of ClusterOn using ~13.5k lines of C++/Python code (counted with CLOC [4]). Table 7.2 summarizes our implementation effort with respect to new or modified code.

Type	Components	LoC	Subtotal
ClusterOn	Core	5,617	
	Events	307	
	Messaging	1,772	
	MQ handler	382	
	Coordinator	1,031	
	Lock server APIs	943	
	Client lib	1,528	11,580
Parsers	Google protobuf	262	
	Redis + SSDB	333	595
Apps	tHT	[966]+107	
	tLog	[966]+286	
	tMT	[966]+98	1457
Total			13632

Table 7.2: ClusterOn LoC breakdown. tHT, tLog, and tMT [11] are built on top of a datalet template [966 LoC] provided by ClusterOn.

Controlet	**MS+SC**	**MS+EC**	**AA+SC**	**AA+EC**
LoC	[150]+191	[150]+37	[150]+62	[150]+38

Table 7.3: LoC breakdown for all pre-built controlets provided by ClusterOn. Pre-built controlets are built on top of the controlet template [150 LoC] provided by ClusterOn.

ClusterOn consists of seven sub-components. (1) *Core* implements the main logic of distributed management of ClusterOn, and constitutes a large portion of the implementation. (2) *Events* uses an event-based approach to achieve request pipelining. This part accounts for around 300 LoC. (3) *Messaging* handles messaging among controlets and datalets. (4) *MQ handler* is used to manage the partitions and offsets for Kafka [21] that we use as our MQ. (5) *Coordinator* consists of two parts: a Python-written failover manager that directly controls the data recovery as well as handling ClusterOn process failover. The coordinator uses ZooKeeper [113] to store topology metadata of the whole cluster and coordinates leader elections during failover. (6) *Lock server APIs* implement client-side logic to acquire and release locks in the remote lock server, which also relies on ZooKeeper for fault tolerance. ClusterOn provides two lock server options—ZooKeeper-based [43] and Redlock-based [32]. (7) *client lib* integrates our Google Protobuf-based protocol into a in-memory KV store client library libmc [22]. We add/modify ~1.5k lines of C++ code.

In order to glue a datalet into the ClusterOn-provided control plane, the developer should provide a protocol parser. We implement three new datalet applications that all share a ClusterOn-defined protocol parser built using Google Protocol Buffers [28] (262 LoC). *tHT* (107 LoC) is a simple in-memory hash table. *tLog* (286 LoC) is a persistent log-structured store that uses *tHT* as the in-memory index. *tMT* (98 LoC) is a datalet based on embedded Masstree [11], which supports range queries. Moreover, in our datalets, ~90% of code (966 LoC) is shared and account for common functionalities such as networking/messaging (NW) and threading/event handling (UT). This code is provided by ClusterOn as the datalet template. Table 7.3 summarizes the number of LoC for the pre-built controlets that are provided by ClusterOn. The four controlets share some events provided by the controlet template, which consist of around 150 LoC.

To further demonstrate the flexibility of ClusterOn, we port two existing standalone and networked KV store applications to ClusterOn—SSDB [37] and Redis [30]. We implemented a simple text-based protocol parser using 333 LoC and integrated it in client library, controlet, and datalet, for both applications.

Overall, it took us less than one person-day to design, develop, and test each datalet (the `Apps` row in Table 7.2) and three person-days on average to implement each controlet (as listed in Table 7.3 excluding the design phase). This underscores ClusterOn's ability to ease development of KV-store-based services.

7.4 Evaluation

In this section, we evaluate ClusterOn's utility in scalability, and performance using a wide range of datalets.

7.4.1 Experimental Setup

Testbeds and configuration We perform our evaluation on Google Cloud Engine (GCE) and a local testbed. For larger scale experiments (§7.4.2), we make use of VMs provisioned from the `us-east1-b` Zone in GCE. Each controlet–datalet pair runs on an `n1-standard-4` VM instance type, which has 4 virtual CPUs and 15 GB memory. Workloads are generated on a separate cluster comprising nodes of `n1-highcpu-8` VM type with 8 virtual CPUs to saturate the cloud network and server-side CPUs. A 1 Gbps network interconnect was used.

For performance stress test (§7.4.3), we use a local testbed consisting of 12 physical machines, each equipped with 8 2.0 GHz Intel Xeon cores, 64 GB memory, with a 10 Gbps network interconnect. The coordinator is a single process (backed-up using ZooKeeper [113] with a standby process as follower) configured to exchange heartbeat messages every 5 sec with controlets. We deploy the coordinator, ZooKeeper, and MQ on a separate set of nodes.

In both cloud and local testbed experiments, we observe that one MQ node is capable of handling at least 8 ClusterOn nodes. For instance, we use a 6-node MQ cluster to sustain the peak throughput of a 48-node ClusterOn cluster in GCE. Both MQ and ClusterOn's coordinator communicate with ZooKeeper for storing metadata and fault tolerance.

Workloads We use workload suites from the Yahoo! Cloud Serving Benchmark (YCSB) [78], a commonly used benchmark for KV stores. Specifically, we present the results with `WorkloadA`, `WorkloadB`, and `WorkloadE`. `WorkloadA` (which we call 50% `Get` in the rest of the section) is an update-intensive workload with a `Get`:`Put` ratio of 50%:50%. `WorkloadB` (which we call 95% `Get`) is a read-mostly workload with 95% `Get`. `WorkloadE` is a scan-intensive workload with 95% `Scan` and 5% `Put`. All workloads consist of 10 million unique KV tuples, each with 16 B key and 32 B value, unless mentioned otherwise. Each benchmark process generates 10 million operations following a balanced uniform KV popularity distribution and a skewed Zipfian distribution (where Zipfian constant = 0.99). The reported throughput is measured in terms of thousand queries per second (kQPS) as an arithmetic mean of three runs.

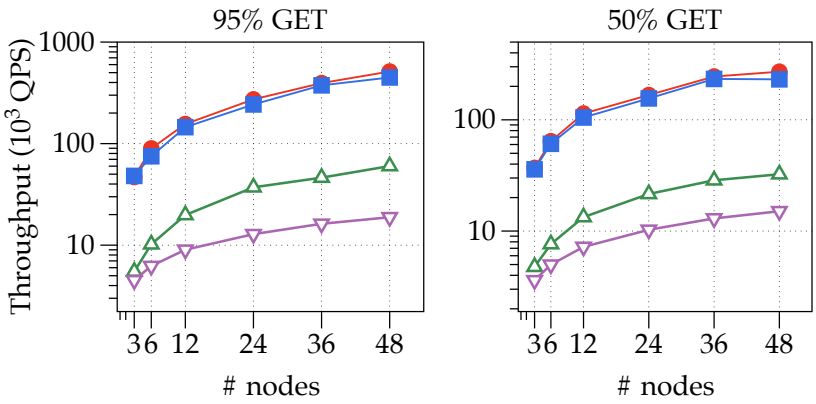
7.4.2 Scalability

Given a non-distributed KV store, ClusterOn promises to automatically scale it out on a cluster of nodes and provide distributed management features. In this test, we evaluate the scalability of the ClusterOn-enabled distributed KV store. We use four single-server KV stores: (1) tHT, and tLog, as examples of newly developed datalets; and (2) tSSDB [37] and tMT [11], as representatives of existing persistent KV stores.

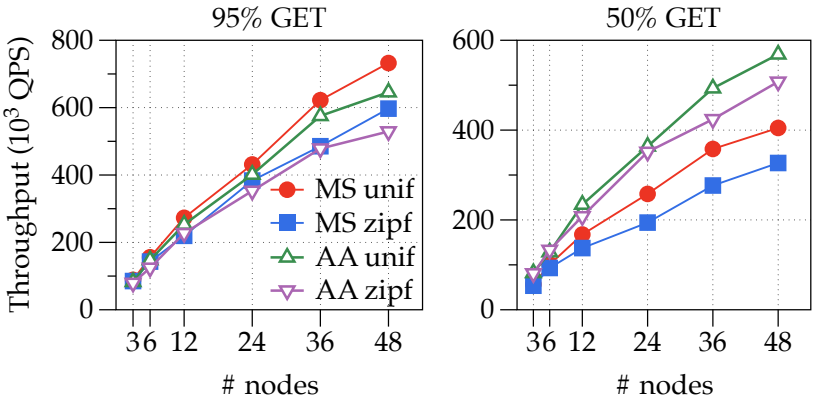
Figure 7.4 shows the scalability of ClusterOn-enabled distributed tHT. We measure the throughput when scaling out tHT from 3 to 48 VMs on GCE. The number of replicas is set to three. We present results for all four topology and consistency combinations: MS+SC, MS+EC, AA+SC, and AA+EC. For all cases, ClusterOn scales tHT out linearly as the number of nodes increases for both read-intensive (95% `Get`) and write-intensive (50% `Get`) workloads. For SC, MS+SC using chain replication scales well, while AA+SC performs worse as expected in locking based implementation. For EC, the results show that our MQ-based EC support scales well for both MS+EC and AA+EC. Performance comparison to existing distributed KV stores will follow in §7.4.3.

7.4.3 Performance Comparison with Natively Distributed Systems

In this experiment, we compare ClusterOn-enabled KV stores with two widely used natively-distributed (off-the-shelf) KV stores: Cassandra [133] and LinkedIn's Voldemort [187]. These experiments were conducted on our 12-node local testbed in order to avoid confounding issues arising from sharing a virtualized platform. We launch the storage servers on six nodes and



(a) Strong Consistency. On log scale.



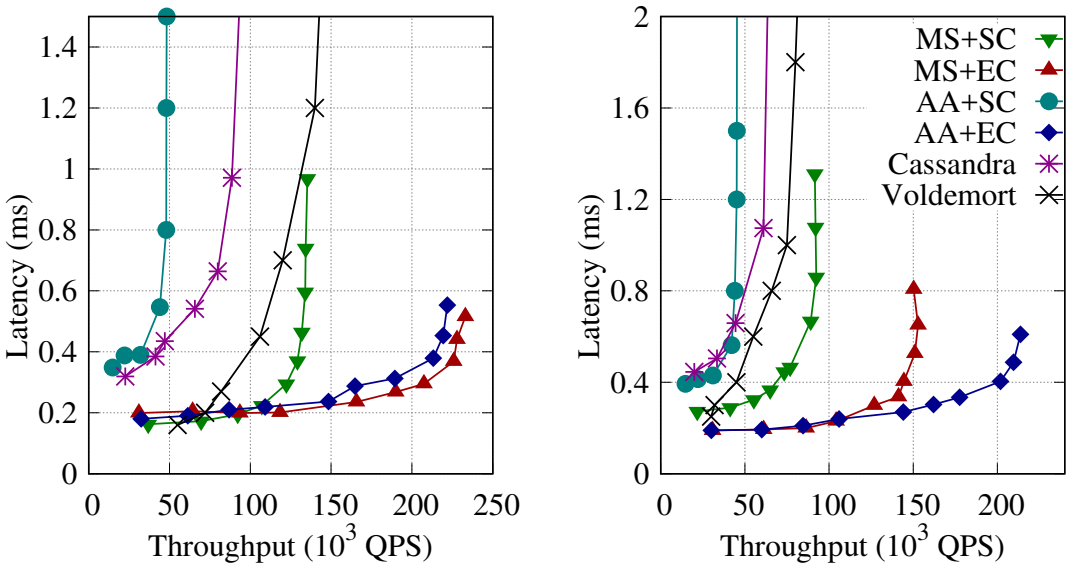
(b) Eventual Consistency.

Figure 7.4: ClusterOn scales *tHT* horizontally.

YCSB clients on the other four nodes to saturate the server side. The coordinator, lock server (only for AA+SC), and ZooKeeper [113] are co-located on a dedicated node, while Kafka (MQ) is run separately on another node. We use tHT as a datalet and show that ClusterOn-enabled KV stores can achieve comparable (sometime better) performance with state-of-the-art systems, proving its high efficiency.

For Cassandra, we specify a read and write consistency level of *one* to make consistency requirements less stringent. Cassandra's replication mechanism follows the AA topology with EC [64]. For Voldemort we use a *server*-side routing policy, *all-routing* as the routing strategy, a replication factor of *three*, *one* as the number of reads or writes that can succeed without the client getting an exception, and persistence set to *memory*.

Figure 7.5 shows the latency and throughput for all tested systems/configurations when



(a) 95% Get. (b) 50% Get.

Figure 7.5: Average latency vs. throughput achieved by various systems under Zipfian workloads. `Dyno`: Dynomite.

varying the number of clients to increase the throughput in units of kQPS.[4] For AA+EC, ClusterOn outperforms Cassandra and Voldemort. For read-intensive workload (95% `Get`), ClusterOn's throughput gain over Cassandra and Voldemort is 4.5× and 1.6×, respectively. For write-intensive workload (50% `Put`), ClusterOn's throughput gain is 4.4× over Cassandra and 2.75× over Voldemort. This is because Cassandra is written in Java and uses compaction in its storage engine. Use of compaction significantly effects the write performance and increases the read latency due to use of extra CPU and disk usage [12]. Similarly, Voldemort is written in Java and uses the same design. Both are based on Amazon's Dynamo paper [83], but the main difference is that Voldemort follows a simple key/value model, while Cassandra uses a persistence model based on BigTable's [65] column-oriented model. Furthermore, our findings are consistent with Dynomite [12] in terms of the performance comparison with Cassandra.

As an extra data point, we also see interesting tradeoffs when experimenting with different configurations supported by ClusterOn. For instance, MS+EC achieves performance comparable to AA+EC under 95% `Get` workload since both configurations serve `Get`s from all replicas. AA+EC achieves 47% higher throughput than MS+EC under 50% `Get` workload, because AA+EC serves `Put`s from all replicas. For AA+SC, lock contention at the DLM caps the performance for both read- and write-intensive workloads. As a result, MS+SC

[4] Uniform workloads show similar trend, hence are omitted.

performs 3.2× better than AA+SC for read-intensive workload and around 2× better for the write-intensive workload.

7.5 Chapter Summary

We have presented the design and implementation of ClusterOn, a framework, which adopts a series of efficient and reusable distributed system components to provide a range of KV store design choices. The framework significantly reduces the effort for developers to extend a single-server KV store or plug existing applications to a highly scalable, fault tolerant, distributed deployment with desired features. Evaluation shows that ClusterOn is flexible, achieves high performance, and scales horizontally on a 48-VM cloud cluster.

Chapter 8

Conclusion and Future Work

A key goal of this dissertation is to make innovative solutions for real-world data management problems. In this dissertation, we have successfully applied the workload-awareness principle and methodology to uncover critical issues in distributed and data-intensive systems. Investigations have led to novel and effective approaches to solve these problems. For example, the offline flash caching heuristic [74] that we developed is, to the best of our knowledge, the first offline caching algorithm that takes flash endurance into consideration. It can be used to evaluate any online flash caching solutions, and navigate the tradeoffs between performance and endurance.

This dissertation is driven by the complexities of modern computing and data-intensive systems, and the need for more efficient and flexible approaches to manage such complexities. The work of this dissertation targets two real-world application scenarios: (1) massive-scale web applications (e.g., Facebook web queries) that require efficient, scalable, and flexible distributed storage systems; (2) cloud-based big data analytics (e.g., distributed NoSQL queries) that need careful deployment planning. By performing extensive and deep analysis to understand the issues [72], designing rigorous models [72, 73] and practical tools [74] to characterize complex workload behaviors, and building efficient systems [52, 70, 71] to manage different tradeoffs and the massive volume of data, this dissertation demonstrates improved efficiency and usability at the system level with a broad focus on practical and user-centric metrics.

8.1 Summary

The extreme latency and throughput requirements of modern web-scale services (an impressive list of users includes Facebook, Airbnb, Twitter, and Wikipedia, etc.) are driving the use of distributed object caches (e.g., Memcached) and stores (e.g., OpenStack Swift). Commonly, the distributed cache/storage tier can scale to hundreds of nodes. With the

growth of cloud platforms and services, object storage solutions have also found their way into both public and private clouds. Cloud service providers such as Amazon and Google Cloud Platform already support object caching and storage as a service. However, web query workloads exhibit high access skew and time varying heterogeneous request patterns, which cause severe access load imbalance and unstable performance. The difficult-to-develop (and -debug) nature of distributed storage systems even makes the situation worse. *This dissertation along this line tackles the above issues using a holistic redesign approach that cohesively combines piece-by-piece optimizations with the goal of maximizing both efficiency and flexibility.*

We first target internet-scale web workloads which now dominates the majority of the network traffic in the world. We apply a novel data partitioning technique to intelligently shard the data at flexibly various granularities. This approach naturally provides workload-awareness by treating hot/cold data differently. Specifically, in memory cache deployment, we developed MBal [71], an in-memory object caching framework that leverages fine-grained data partitioning and adaptive multi-phase load balancing. MBal performs fast, lockless inserts (SET) and lookups (GET) by partitioning user objects and compute/memory resources into non-overlapping subsets called cachelets. It quickly detects presence of hotspots in the workloads and uses an adaptive, multi-phase load balancing approach to mitigate any load imbalance. The cachelet-based design of MBal provides a natural abstraction for object migration both within a server and across servers in a cohesive manner. Evaluation results shows that MBal brings down the tail latency of imbalanced web queries close to that of an ideally balanced workload.

Second, we look at data placement problem in public cloud environments. At a very high level, this work adds a smart management tier on top of the cloud storage service layer by exploiting workload and storage heterogeneity (i.e., workload-awareness). The use of cloud resources frees tenants from the traditionally cumbersome IT infrastructure planning and maintenance, and allows them to focus on application development and optimal resource deployment. These desirable features coupled with the advances in virtualization infrastructure are driving the adoption of public, private, and hybrid clouds for not only web applications, such as Netflix, Instagram and Airbnb, but also modern big data analytics using parallel programming paradigms such as Hadoop and Dryad. With the improvement in network connectivity and emergence of new data sources such as IoT edge points, mobile platforms, and wearable devices, enterprise-scale data-intensive analytics now involves terabyte- to petabyte-scale data with more data being generated from these sources constantly. Thus, storage allocation and management would play a key role in overall performance improvement and cost reduction for this domain. While cloud makes data analytics easy to deploy and scale, the vast variety of available storage services with different persistence, performance and capacity characteristics, presents unique challenges for deploying big data analytics in the cloud. *The proposed cloud storage tiering solution, driven by real-world workload behaviors and cloud service characterization, takes the first step towards providing cost-effective data placement support for cloud-based big data analytics using the economic*

principles of demand and supply for both cloud tenants and service providers. While the use of faster storage devices such as SSDs is desirable by tenants, it incurs significant maintenance costs to the cloud service provider. To alleviate this problem, we extended CAST to further incorporate dynamic pricing by involving the provider. The resulting hybrid cloud store [73] exposes the storage tiering to tenants with a dynamic pricing model that is based on the tenants' usage and the provider's desire to maximize profit. The tenants leverage knowledge of their workloads and current pricing information to select a data placement strategy that would meet the application requirements at the lowest cost. Our approach allows both a service provider and its tenants to engage in a pricing game, which yields a win-–win situation, as shown in the results of the production trace driven simulations.

This dissertation then tackles the local storage data management issues by focusing on local flash cache heuristic optimizations. Local storage serves as the fundamental building block for almost all possible high-level storage applications such as relational databases, key-value stores, etc. This work provides a generic solution for the example applications mentioned in the first two works covered in this dissertation. Unlike traditional hard disk drivers (HDDs), flash drives, e.g., NAND-based solid state drives (SSDs), have limits on endurance (i.e., the number of times data can be erased and overwritten). Furthermore, the unit of erasure can be many times larger than the basic unit of I/Os, and this leads to complexity with respect to consolidating live data and erasing obsolete data. For enterprise primary storage workloads, storage must balance the requirement for large capacity, high performance, and low cost. A well studied technique is to place a flash cache in front of larger, HDD-based storage system, which strives to achieve the performance benefit of SSD devices and the low cost per GB efficiency of HDD devices. In this scenario, the choice of a cache replacement algorithm can make a significant difference in both performance and endurance. While there are many cache replacement algorithms, their effectiveness is hard to judge due to the lack of a baseline against which to compare them: Belady's MIN, the usual offline best-case algorithm, considers read hit ratio but not endurance. To this end, we explored offline algorithms for flash caching in terms of both hit ratio and flash lifespan. We developed a multi-stage heuristic [74] by synthesizing several techniques that manage data at the granularity of a flash erasure unit (which we call a container) to approximate the offline optimal algorithm, which we believe is much harder to compute. Evaluation showed that the container-optimized offline heuristic provides the optimal read hit ratio as MIN with 67% less flash erasures. More fundamentally, my investigation provides a useful approximate baseline for evaluating any online algorithm, highlighting the importance of comparing new policies for caching compound blocks in flash.

Finally, this dissertation answers a fundamental research problem–how to build a fully-functional distributed key-value storage system with minimal engineering effort. This work is motivated by the real demand that real-world developers are faced with when building a complex distributed system from scratch. As is known, distributed systems are notoriously bug-prone and difficult to implement. What would be an easy and productive way to develop distributed systems from scratch? To solve the research problem, we conducted a study [53],

where we observed that, to a large extent, such systems would implement their own way of handling features of replication, fault tolerance, consistency, and cluster topology, etc. To this end, we designed and implemented ClusterOn, a universal and flexible ecosystem that handles the "messy plumbings" of distributed systems by synthesizing a series of reusable and efficient components of distributed system techniques. Using ClusterOn, developers only need to focus on the core function (SET or GET logics) implementation of the application, and ClusterOn will convert it into a scalable, and highly configurable distributed deployment, following a serverless fashion.

8.2 Future Directions

This dissertation are focused on practical problems that exist in storage systems. We are particularly interested in designing systems with high efficiency/flexibility and better security, and extend our understanding in cyber-physical systems and ubiquitous computing. In the following, We discuss several future directions as an extension to this dissertation.

8.2.1 Redesign the System-Architecture Interfaces

Massive deployment of fast storage devices (such as high-density non-volatile memory) has boosted the performance of modern datacenter applications. The prior and ongoing work has shown that there exists great potential for extending the endurance of datacenter flash storage in both single device [74] and at scale. At extreme scale, performing distributed wear leveling and global garbage collecting can not only improve the overall lifespan of the flash array, but also effectively improve the overall performance. However, applications and the underlying distributed flash cluster are still segregated and there is no way application can directly manage the functionality controlled by the storage hardware, thus impacting the cost effectiveness inevitably. Observations and preliminary results demonstrate the huge improvement space of such an application-managed hardware design [74]. In the short term, we plan to conduct research for a cross-layer system-architecture codesign to enable transparent scale-up/scale-out high performance storage.

8.2.2 Rethinking System Software Design in Ubiquitous Computing Era

Computing has expanded beyond the Internet and become ubiquitous everywhere in the physical world. A trending area is rethinking the server system design in the context of cyber-physical systems, IoT, mobile and wearable devices. Future server systems would involve complex interactions with users and require the right balance of resources such as CPU,

memory, storage, and energy. To provide effective and efficient infrastructure support, We are interested in exploring the tradeoffs among all possible objectives including performance, deployment cost, reliability, easy-of-use/deployment, and energy efficiency, in the context of low-end, low-capacity, energy-sensitive and IoT-scale environments. Such studies will impact the next-generation server system software design and development, and provide best practice guidance for practical deployment in the field.

8.2.3 Utilizing Heterogeneous Accelerators for Better HPC I/O

As large-scaled infrastructure keeps to exploit heterogenous accelerators, fully utilization of both computing power and associative memory systems is of greater importance [105, 108, 109, 110]. To make programs more efficiently run on modern commodity server systems, we need to fully exploit the parallel potentials that exists in the machines. It was known that parallel accelerators, such as GPUs, require careful usages of different memory levels for performance improvements [107, 194], and data-level parallelism can help better utilize memory bandwidth [106, 206]. The problem becomes more exciting when considering a HPC accelerator I/O system codesign. In the short term, I plan to (1) study and understand the behaviors of utilizing heterogeneous accelerators in HPC I/O subsystems, and (2) build next-generation infrastructure support that aims to transparently exploiting the parallelism of large-scale heterogeneous accelerators to enhance and boost the I/O performance and energy efficiency of HPC I/O systems and HPC storage hardware.

8.2.4 Security in Next-Generation System Infrastructure

Mining security risks are becoming increasingly challenging as the amount of data grows rapidly. New techniques [44, 61, 69, 154, 155, 185] are needed to detect, prioritize and measure the security risks from massive datasets. From this angle, in the long term, we are particularly interested in utilizing data mining and machine learning techniques to effectively enhance the security of next-generation system infrastructure such as IoT, mobile networks, and datacenters in the edge. Specific directions include conducting empirical analysis on open platforms such as Android App Store, Facebook etc. for better understanding of existing or potential security issues, and system software design and implementation on top of emerging hardware- or architecture-level techniques such as Intel SGX [34].

Bibliography

[1] Azure Storage. http://goo.gl/fcSbYb.

[2] OpenStack Ceilometer. https://wiki.openstack.org/wiki/Ceilometer.

[3] Ceph Storage Pools. http://docs.ceph.com/docs/argonaut/config-cluster/pools/.

[4] Count Lines of Code (CLOC). http://cloc.sourceforge.net/.

[5] Cloud Resource Allocation Games. https://www.ideals.illinois.edu/handle/2142/17427.

[6] Advanced Engineering Mathematics. http://goo.gl/wyHaFb.

[7] Amazon EC2 Pricing. http://aws.amazon.com/ec2/pricing/, .

[8] EC2 Storage. http://goo.gl/L7nnEy, .

[9] Amazon Web Service ElastiCache. http://aws.amazon.com/elasticache/, .

[10] Amazon Elastic MR. http://goo.gl/GXzwaU, .

[11] Embedded Masstree. https://github.com/rmind/masstree.

[12] Why not Cassandra. http://www.dynomitedb.com/docs/dynomite/v0.5.6/faq/.

[13] Google Cloud Storage Connector for Hadoop. http://goo.gl/ji4qqp, .

[14] Google Compute Engine Pricing. https://cloud.google.com/compute/#pricing, .

[15] Hadoop on Google Compute Engine. http://goo.gl/6tf3vg.

[16] Apache Hadoop. http://hadoop.apache.org/.

[17] HDFS-2832. http://goo.gl/j9rDwi.

[18] Microsoft Azure HDinsight. http://goo.gl/wsQ9QG.

[19] HP Cloud Storage. http://goo.gl/BjmwFF.

[20] Impala. http://impala.io/.

[21] Apache Kafka. http://kafka.apache.org/.

[22] libmc, . https://github.com/douban/libmc.

[23] libmemcached. http://libmemcached.org/, .

[24] Memcached. http://memcached.org/.

[25] Mumak: MapReduce Simulator. http://goo.gl/AR6BeV.

[26] Apache Oozie. http://oozie.apache.org/.

[27] OpenStack Swift Policies. http://docs.openstack.org/developer/swift/overview_policies.html.

[28] Google Protocol Buffers. https://developers.google.com/protocol-buffers/.

[29] pylibmc. https://pypi.python.org/pypi/pylibmc.

[30] Redis. http://redis.io/.

[31] Redis. http://redis.io/, .

[32] Distributed locks with Redis, . http://redis.io/topics/distlock.

[33] S3 as a replacement of HDFS. https://wiki.apache.org/hadoop/AmazonS3.

[34] Intel SGX. https://software.intel.com/en-us/sgx.

[35] SpyMemcached. https://code.google.com/p/spymemcached/.

[36] SSD vs. HDD Pricing: Seven Myths That Need Correcting. http://www.enterprisestorageforum.com/storage-hardware/ssd-vs.-hdd-pricing-seven-myths-that-need-correcting.html, .

[37] SSDB, . https://github.com/ideawu/ssdb.

[38] Swift Connector to Hadoop. http://docs.openstack.org/developer/sahara/userdoc/hadoop-swift.html.

[39] Cache with Twemcache. https://blog.twitter.com/2012/caching-with-twemcache.

[40] How Twitter Uses Redis to Scale. http://highscalability.com/blog/2014/9/8/how-twitter-uses-redis-to-scale.html.

[41] Memcached Protocol. https://code.google.com/p/memcached/wiki/NewProtocols.

[42] Scaling Memcached with vBuckets. http://dustin.sallings.org/2010/06/29/memcached-vbuckets.html.

[43] ZooKeeper Recipes and Solutions. https://zookeeper.apache.org/doc/r3.1.2/recipes.html.

[44] Measuring the insecurity of mobile deep links. In *26th USENIX Security Symposium (USENIX Security 17)*, Vancouver, BC, 2017. USENIX Association. URL https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/liu.

[45] Anurag Acharya, Mustafa Uysal, and Joel Saltz. Active disks: Programming model, algorithms and evaluation. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VIII, pages 81–91, New York, NY, USA, 1998. ACM. ISBN 1-58113-107-0. doi: 10.1145/291069.291026. URL http://doi.acm.org/10.1145/291069.291026.

[46] Orna Agmon Ben-Yehuda, Eyal Posener, Muli Ben-Yehuda, Assaf Schuster, and Ahuva Mu'alem. Ginseng: Market-driven memory allocation. In *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '14, pages 41–52, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2764-0. doi: 10.1145/2576195.2576197. URL http://doi.acm.org/10.1145/2576195.2576197.

[47] Susanne Albers, Sanjeev Arora, and Sanjeev Khanna. Page replacement for general caching problems. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA'99)*, 1999.

[48] Christoph Albrecht, Arif Merchant, Murray Stokely, Muhammad Waliji, François Labelle, Nate Coehlo, Xudong Shi, and C. Eric Schrock. Janus: Optimal flash provisioning for cloud storage workloads. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 91–102, San Jose, CA, 2013. USENIX. ISBN 978-1-931971-01-0. URL https://www.usenix.org/conference/atc13/technical-sessions/presentation/albrecht.

[49] Ganesh Ananthanarayanan, Ali Ghodsi, Andrew Warfield, Dhruba Borthakur, Srikanth Kandula, Scott Shenker, and Ion Stoica. Pacman: Coordinated memory caching for parallel jobs. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 267–280, San Jose, CA, 2012. USENIX. ISBN 978-931971-92-8. URL https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/ananthanarayanan.

[50] A. Anwar, Y. Cheng, and A. R. Butt. Towards managing variability in the cloud. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 1081–1084, May 2016. doi: 10.1109/IPDPSW.2016.62.

[51] Ali Anwar, Yue Cheng, Aayush Gupta, and Ali R. Butt. Taming the cloud object storage with mos. In *Proceedings of the 10th Parallel Data Storage Workshop*, PDSW '15, pages 7–12, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-4008-3. doi: 10.1145/2834976.2834980. URL http://doi.acm.org/10.1145/2834976.2834980.

[52] Ali Anwar, Yue Cheng, Aayush Gupta, and Ali R. Butt. Mos: Workload-aware elasticity for cloud object stores. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '16, pages 177–188, New York, NY, USA, June 2016. ACM. ISBN 978-1-4503-4314-5. doi: 10.1145/2907294.2907304. URL http://doi.acm.org/10.1145/2907294.2907304.

[53] Ali Anwar, Yue Cheng, Hai Huang, and Ali R. Butt. Clusteron: Building highly configurable and reusable clustered data services using simple data nodes. In *8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 16)*, Denver, CO, June 2016. USENIX Association. URL https://www.usenix.org/conference/hotstorage16/workshop-program/presentation/anwar.

[54] Raja Appuswamy, Christos Gkantsidis, Dushyanth Narayanan, Orion Hodson, and Antony Rowstron. Scale-up vs scale-out for hadoop: Time to rethink? In *Proceedings of ACM SoCC 2013*. ISBN 978-1-4503-2428-1. doi: 10.1145/2523616.2523629. URL http://doi.acm.org/10.1145/2523616.2523629.

[55] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, pages 53–64, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1097-0. doi: 10.1145/2254756.2254766. URL http://doi.acm.org/10.1145/2254756.2254766.

[56] Shivnath Babu. Towards automatic optimization of mapreduce programs. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 137–142, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0036-0. doi: 10.1145/1807128.1807150. URL http://doi.acm.org/10.1145/1807128.1807150.

[57] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Syst. J.*, 5(2):78–101, June 1966. ISSN 0018-8670. doi: 10.1147/sj.52.0078. URL http://dx.doi.org/10.1147/sj.52.0078.

[58] Avraham Ben-Aroya and Sivan Toledo. Competitive analysis of flash memory algorithms. *ACM Trans. Algorithms*, 7(2):1–37, March 2011. ISSN 1549-6325. doi: 10.1145/1921659.1921669. URL http://doi.acm.org/10.1145/1921659.1921669.

[59] M. Berezecki, E. Frachtenberg, M. Paleczny, and K. Steele. Many-core key-value store. In *Green Computing Conference and Workshops (IGCC), 2011 International*, pages 1–8, July 2011. doi: 10.1109/IGCC.2011.6008565.

[60] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IX, pages 117–128, New York, NY, USA, 2000. ACM. ISBN 1-58113-317-0. doi: 10.1145/378993.379232. URL http://doi.acm.org/10.1145/378993.379232.

[61] Amiangshu Bosu, Fang Liu, Danfeng (Daphne) Yao, and Gang Wang. Collusive data leak and more: Large-scale threat analysis of inter-app communications. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '17, pages 71–85, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4944-4. doi: 10.1145/3052973.3053004. URL http://doi.acm.org/10.1145/3052973.3053004.

[62] M. Brehob, S. Wagner, E. Torng, and R. Enbody. Optimal replacement is np-hard for nonstandard caches. *Computers, IEEE Transactions on*, 53(1):73–76, Jan 2004. ISSN 0018-9340. doi: 10.1109/TC.2004.1255792.

[63] Ali R. Butt, Chris Gniady, and Y. Charlie Hu. The performance impact of kernel prefetching on buffer cache replacement algorithms. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'05)*, 2005. ISBN 1-59593-022-1. doi: 10.1145/1064212.1064231. URL http://doi.acm.org/10.1145/1064212.1064231.

[64] Neville Carvalho, Hyojun Kim, Maohua Lu, Prasenjit Sarkar, Rohit Shekhar, Tarun Thakur, Pin Zhou, and Remzi H Arpaci-Dusseau. Finding consistency in an inconsistent world: Towards deep semantic understanding of scale-out distributed databases. In *USENIX HotStorage '16*.

[65] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. *Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[66] H. C. Chang, B. Li, M. Grove, and K. W. Cameron. How processor speedups can slow down i/o performance. In *2014 IEEE 22nd International Symposium on Modelling, Analysis Simulation of Computer and Telecommunication Systems*, pages 395–404, Sept 2014. doi: 10.1109/MASCOTS.2014.55.

[67] H. C. Chang, B. Li, G. Back, A. R. Butt, and K. W. Cameron. Luc: Limiting the unintended consequences of power scaling on parallel transaction-oriented workloads.

In *2015 IEEE International Parallel and Distributed Processing Symposium*, pages 324–333, May 2015. doi: 10.1109/IPDPS.2015.99.

[68] Yanpei Chen, Sara Alspaugh, and Randy Katz. Interactive analytical processing in big data systems: A cross-industry study of mapreduce workloads. *Proc. VLDB Endow.*, 5 (12):1802–1813, August 2012. ISSN 2150-8097. doi: 10.14778/2367502.2367519. URL http://dx.doi.org/10.14778/2367502.2367519.

[69] Long Cheng, Fang Liu, and Danfeng (Daphne) Yao. Enterprise data breach: causes, challenges, prevention, and future directions. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, pages n/a–n/a. ISSN 1942-4795. doi: 10.1002/widm.1211. URL http://dx.doi.org/10.1002/widm.1211.

[70] Yue Cheng, Aayush Gupta, Anna Povzner, and Ali R. Butt. High performance in-memory caching through flexible fine-grained services. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 56:1–56:2, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2428-1. doi: 10.1145/2523616.2525964. URL http://doi.acm.org/10.1145/2523616.2525964.

[71] Yue Cheng, Aayush Gupta, and Ali R. Butt. An in-memory object caching framework with adaptive load balancing. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, pages 4:1–4:16, New York, NY, USA, April 2015. ACM. ISBN 978-1-4503-3238-5. doi: 10.1145/2741948.2741967. URL http://doi.acm.org/10.1145/2741948.2741967.

[72] Yue Cheng, M. Safdar Iqbal, Aayush Gupta, and Ali R. Butt. Cast: Tiering storage for data analytics in the cloud. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '15, pages 45–56, New York, NY, USA, June 2015. ACM. ISBN 978-1-4503-3550-8. doi: 10.1145/2749246.2749252. URL http://doi.acm.org/10.1145/2749246.2749252.

[73] Yue Cheng, M. Safdar Iqbal, Aayush Gupta, and Ali R. Butt. Pricing games for hybrid object stores in the cloud: Provider vs. tenant. In *7th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 15)*, Santa Clara, CA, July 2015. USENIX Association. URL https://www.usenix.org/conference/hotcloud15/workshop-program/presentation/cheng.

[74] Yue Cheng, Fred Douglis, Philip Shilane, Michael Trachtman, Grant Wallace, Peter Desnoyers, and Kai Li. Erasing belady's limitations: In search of flash cache offline optimality. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, Denver, CO, June 2016. USENIX Association. URL https://www.usenix.org/conference/atc16/technical-sessions/presentation/cheng.

[75] Yue Cheng, M. Safdar Iqbal, Aayush Gupta, and Ali R. Butt. Provider versus tenant pricing games for hybrid object stores in the cloud. *IEEE Internet Computing*, 20(3): 28–35, 2016. ISSN 1089-7801. doi: doi.ieeecomputersociety.org/10.1109/MIC.2016.50.

[76] Marek Chrobak, Gerhard J. Woeginger, Kazuhisa Makino, and Haifeng Xu. Caching is hard—even in the fault model. *Algorithmica*, 63(4):781–794, 2012. ISSN 0178-4617. doi: 10.1007/s00453-011-9502-9. URL http://dx.doi.org/10.1007/s00453-011-9502-9.

[77] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0036-0. doi: 10.1145/1807128.1807152. URL http://doi.acm.org/10.1145/1807128.1807152.

[78] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0036-0. doi: 10.1145/1807128.1807152. URL http://doi.acm.org/10.1145/1807128.1807152.

[79] Couchbase. vbuckets: The core enabling mechanism for couchbase server data distribution (aka "auto-sharding"). Technical Whitepaper.

[80] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with cfs. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01, pages 202–215, New York, NY, USA, 2001. ACM. ISBN 1-58113-389-8. doi: 10.1145/502034.502054. URL http://doi.acm.org/10.1145/502034.502054.

[81] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1251254.1251264.

[82] Biplob Debnath, Sudipta Sengupta, and Jin Li. Flashstore: High throughput persistent key-value store. *Proceedings of the VLDB Endowment*, 3(1-2):1414–1425, September 2010. ISSN 2150-8097. doi: 10.14778/1920841.1921015. URL http://dx.doi.org/10.14778/1920841.1921015.

[83] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. ACM SOSP '07.

[84] Iman Elghandour and Ashraf Aboulnaga. Restore: Reusing results of mapreduce jobs. *Proc. VLDB Endow.*, 5(6):586–597, February 2012. ISSN 2150-8097. doi: 10.14778/2168651.2168659. URL http://dx.doi.org/10.14778/2168651.2168659.

[85] Jason Evans. A scalable concurrent malloc(3) implementation for freebsd. In *BSDCAN'06*, 2006.

[86] Facebook. Facebook Flashcache. https://github.com/facebook/flashcache.

[87] Bin Fan, Hyeontaek Lim, David G. Andersen, and Michael Kaminsky. Small cache, big effect: Provable load balancing for randomly partitioned cluster services. In *Proceedings of the 2Nd ACM Symposium on Cloud Computing*, SOCC '11, pages 23:1–23:12, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0976-9. doi: 10.1145/2038916.2038939. URL http://doi.acm.org/10.1145/2038916.2038939.

[88] Bin Fan, David G. Andersen, and Michael Kaminsky. Memc3: Compact and concurrent memcache with dumber caching and smarter hashing. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, nsdi'13, pages 371–384, Berkeley, CA, USA, 2013. USENIX Association. URL http://dl.acm.org/citation.cfm?id=2482626.2482662.

[89] Ziqi Fan, David HC Du, and Doug Voigt. H-arc: A non-volatile memory based cache policy for solid state drives. In *Proceedings of the 30th Mass Storage Systems and Technologies Symposium (MSST'14)*. IEEE, 2014.

[90] Rohan Gandhi, Aayush Gupta, Anna Povzner, Wendy Belluomini, and Tim Kaldewey. Mercury: Bringing efficiency to key-value stores. In *Proceedings of the 6th International Systems and Storage Conference*, SYSTOR '13, pages 6:1–6:6, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2116-7. doi: 10.1145/2485732.2485746. URL http://doi.acm.org/10.1145/2485732.2485746.

[91] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *ACM SOSP'03*.

[92] Binny S. Gill. On multi-level exclusive caching: Offline optimality and why promotions are better than demotions. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST'08)*, February 2008. URL http://dl.acm.org/citation.cfm?id=1364813.1364817.

[93] Chris Gniady, Ali R. Butt, and Y. Charlie Hu. Program-counter-based pattern classification in buffer caching. In *Proceedings of the 6th USENIX Conference on Symposium on Operating Systems Design & Implementation (OSDI'04)*, 2004. URL http://dl.acm.org/citation.cfm?id=1251254.1251281.

[94] GNU Parallel. GNU Parallel. http://www.gnu.org/software/parallel/.

[95] B. Godfrey, K. Lakshminarayanan, S. Surana, R. Karp, and I. Stoica. Load balancing in dynamic structured p2p systems. In *INFOCOM 2004. Twenty-third AnnualJoint*

Conference of the IEEE Computer and Communications Societies, volume 4, pages 2253–2262 vol.4, March 2004. doi: 10.1109/INFCOM.2004.1354648.

[96] Laura M. Grupp, John D. Davis, and Steven Swanson. The bleak future of nand flash memory. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, FAST'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association. URL http://dl.acm.org/citation.cfm?id=2208461.2208463.

[97] Jorge Guerra, Himabindu Pucha, Joseph Glider, Wendy Belluomini, and Raju Rangaswami. Cost effective storage using extent based dynamic tiering. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies*, FAST'11, pages 20–20, Berkeley, CA, USA, 2011. USENIX Association. ISBN 978-1-931971-82-9. URL http://dl.acm.org/citation.cfm?id=1960475.1960495.

[98] Tyler Harter, Dhruba Borthakur, Siying Dong, Amitanand Aiyer, Liyin Tang, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Analysis of hdfs under hbase: A facebook messages case study. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST 14)*, pages 199–212, Santa Clara, CA, 2014. USENIX. ISBN ISBN 978-1-931971-08-9. URL https://www.usenix.org/conference/fast14/technical-sessions/presentation/harter.

[99] Herodotos Herodotou and Shivnath Babu. Profiling, what-if analysis, and cost-based optimization of mapreduce programs, 2011.

[100] Herodotos Herodotou, Harold Lim, Gang Luo, Nedyalko Borisov, Liang Dong, Fatma Bilgen Cetin, and Shivnath Babu. Starfish: A self-tuning system for big data analytics. In *In CIDR*, pages 261–272. ACM, 2011.

[101] David A. Holland, Elaine Angelino, Gideon Wald, and Margo I. Seltzer. Flash caching on the storage client. In *Proceedings of the USENIX Annual Technical Conference (ATC'13)*, 2013. ISBN 978-1-931971-01-0. URL https://www.usenix.org/conference/atc13/technical-sessions/presentation/holland.

[102] Yu-Ju Hong and Mithuna Thottethodi. Understanding and mitigating the impact of load imbalance in the memory caching tier. In *ACM SOCC'13*.

[103] Yu-Ju Hong and Mithuna Thottethodi. Understanding and mitigating the impact of load imbalance in the memory caching tier. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 13:1–13:17, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2428-1. doi: 10.1145/2523616.2525970. URL http://doi.acm.org/10.1145/2523616.2525970.

[104] L. P. Horwitz, R. M. Karp, R. E. Miller, and S. Winograd. Index register allocation. *J. ACM*, 13(1):43–61, January 1966. ISSN 0004-5411. doi: 10.1145/321312.321317. URL http://doi.acm.org/10.1145/321312.321317.

[105] K. Hou, H. Wang, and W. c. Feng. Delivering Parallel Programmability to the Masses via the Intel MIC Ecosystem: A Case Study. In *2014 43rd International Conference on Parallel Processing Workshops*, pages 273–282, Sept 2014.

[106] K. Hou, H. Wang, and W. C. Feng. AAlign: A SIMD Framework for Pairwise Sequence Alignment on x86-Based Multi-and Many-Core Processors. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 780–789, May 2016.

[107] K. Hou, W. c. Feng, and S. Che. Auto-tuning strategies for parallelizing sparse matrix-vector (spmv) multiplication on multi- and many-core processors. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, May 2017.

[108] Kaixi Hou, Hao Wang, and Wu-chun Feng. ASPaS: A Framework for Automatic SIMDization of Parallel Sorting on x86-based Many-core Processors. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, ICS '15, pages 383–392, New York, NY, USA, 2015. ACM.

[109] Kaixi Hou, Weifeng Liu, Hao Wang, and Wu-chun Feng. Fast Segmented Sort on GPUs. In *Proceedings of the 2017 International Conference on Supercomputing*, ICS '17. ACM, 2017.

[110] Kaixi Hou, Hao Wang, and Wu-chun Feng. Gpu-unicache: Automatic code generation of spatial blocking for stencils on gpus. In *Proceedings of the ACM Conference on Computing Frontiers*, CF '17. ACM, 2017.

[111] Qi Huang, Helga Gudmundsdottir, Ymir Vigfusson, Daniel A. Freedman, Ken Birman, and Robbert van Renesse. Characterizing load imbalance in real-world networked caches. In *Proceedings of the 13th ACM Workshop on Hot Topics in Networks*, HotNets-XIII, pages 8:1–8:7, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3256-9. doi: 10.1145/2670518.2673882. URL http://doi.acm.org/10.1145/2670518.2673882.

[112] Sai Huang, Qingsong Wei, Jianxi Chen, Cheng Chen, and Dan Feng. Improving flash-based disk cache with lazy adaptive replacement. In *Proceedings of the IEEE 29th Symposium on Mass Storage Systems and Technologies (MSST'13)*, May 2013. doi: 10.1109/MSST.2013.6558447.

[113] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX ATC'10*.

[114] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, pages 11–11, Berkeley, CA, USA, 2010. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1855840.1855851.

[115] Jinho Hwang and Timothy Wood. Adaptive performance-aware distributed memory caching. In *Proceedings of the 10th International Conference on Autonomic Computing (ICAC 13)*, pages 33–43, San Jose, CA, 2013. USENIX. ISBN 978-1-931971-02-7. URL https://www.usenix.org/conference/icac13/technical-sessions/presentation/hwang.

[116] Jinho Hwang, K. K. Ramakrishnan, and Timothy Wood. Netvm: High performance and flexible networking using virtualization on commodity platforms. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 445–458, Seattle, WA, April 2014. USENIX Association. ISBN 978-1-931971-09-6. URL https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/hwang.

[117] Intel Corporation. Intel data plane development kit: Getting started guide.

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

[119] Virajith Jalaparti, Hitesh Ballani, Paolo Costa, Thomas Karagiannis, and Ant Rowstron. Bridging the tenant-provider gap in cloud services. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, pages 10:1–10:14, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1761-0. doi: 10.1145/2391229.2391239. URL http://doi.acm.org/10.1145/2391229.2391239.

[120] EunYoung Jeong, Shinae Wood, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. mtcp: a highly scalable user-level tcp stack for multicore systems. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 489–502, Seattle, WA, April 2014. USENIX Association. ISBN 978-1-931971-09-6. URL https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/jeong.

[121] Jffs2. Jffs2: The journalling flash file system, version 2. https://sourceware.org/jffs2/.

[122] Rishi Kapoor, George Porter, Malveeka Tewari, Geoffrey M. Voelker, and Amin Vahdat. Chronos: Predictable low latency for data center applications. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, pages 9:1–9:14, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1761-0. doi: 10.1145/2391229.2391238. URL http://doi.acm.org/10.1145/2391229.2391238.

[123] Ramakrishna Karedla, J Spencer Love, and Bradley G Wherry. Caching strategies to improve disk system performance. *Computer*, 27(3):38–46, 1994.

[124] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *ACM STOC '97*, 1997.

[125] David R. Karger and Matthias Ruhl. Simple efficient load balancing algorithms for peer-to-peer systems. In *Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '04, pages 36–43, New York, NY, USA, 2004. ACM. ISBN 1-58113-840-7. doi: 10.1145/1007912.1007919. URL http://doi.acm.org/10.1145/1007912.1007919.

[126] S. Kavalanekar, B. Worthington, Qi Zhang, and V. Sharda. Characterization of storage workload traces from production windows servers. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC'08)*, Sept 2008. doi: 10.1109/IISWC.2008.4636097.

[127] Hyojun Kim, Sangeetha Seshadri, Clement L. Dickey, and Lawrence Chiu. Evaluating phase change memory for enterprise storage systems: A study of caching and tiering approaches. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST 14)*, pages 33–45, Santa Clara, CA, 2014. USENIX. ISBN ISBN 978-1-931971-08-9. URL https://www.usenix.org/conference/fast14/technical-sessions/presentation/kim.

[128] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *SCIENCE*, 220(4598):671–680, 1983.

[129] Avi Kivity, Dor Laor, Glauber Costa, Pekka Enberg, Nadav Har'El, Don Marti, and Vlad Zolotarov. Osv—optimizing the operating system for virtual machines. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 61–72, Philadelphia, PA, June 2014. USENIX Association. ISBN 978-1-931971-10-2. URL https://www.usenix.org/conference/atc14/technical-sessions/presentation/kivity.

[130] Ricardo Koller, Leonardo Marmol, Raju Rangaswami, Swaminathan Sundararaman, Nisha Talagala, and Ming Zhao. Write policies for host-side flash caches. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST'13)*, February 2013. ISBN 978-1-931971-99-7. URL https://www.usenix.org/conference/fast13/technical-sessions/presentation/koller.

[131] Krish K.R., Ali Anwar, and Ali R. Butt. phisched: A heterogenety-aware hadoop workflow scheduler. In *Mascots 2014*. IEEE.

[132] K.R. Krish, A. Anwar, and A.R. Butt. hats: A heterogeneity-aware tiered storage for hadoop. In *Cluster, Cloud and Grid Computing (CCGrid), 2014 14th IEEE/ACM International Symposium on*, pages 502–511, May 2014. doi: 10.1109/CCGrid.2014.51.

[133] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 2010.

[134] Changman Lee, Dongho Sim, Jooyoung Hwang, and Sangyeun Cho. F2fs: A new file system for flash storage. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST'15)*, February 2015. ISBN 978-1-931971-201. URL https://www.usenix.org/conference/fast15/technical-sessions/presentation/lee.

[135] Sungjin Lee, Ming Liu, Sangwoo Jun, Shuotao Xu, Jihong Kim, and Arvind. Application-managed flash. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST'16)*, February 2016. URL http://usenix.org/conference/fast16/technical-sessions/presentation/lee.

[136] B. Li, S. Song, I. Bezakova, and K. W. Cameron. Energy-aware replica selection for data-intensive services in cloud. In *2012 IEEE 20th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 504–506, Aug 2012. doi: 10.1109/MASCOTS.2012.66.

[137] B. Li, S. L. Song, I. Bezakova, and K. W. Cameron. Edr: An energy-aware runtime load distribution system for data-intensive applications in the cloud. In *2013 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 1–8, Sept 2013. doi: 10.1109/CLUSTER.2013.6702674.

[138] B. Li, H. C. Chang, S. Song, C. Y. Su, T. Meyer, J. Mooring, and K. W. Cameron. The power-performance tradeoffs of the intel xeon phi on hpc applications. In *2014 IEEE International Parallel Distributed Processing Symposium Workshops*, pages 1448–1456, May 2014. doi: 10.1109/IPDPSW.2014.162.

[139] Bo Li, Edgar A. León, and Kirk W. Cameron. Cos: A parallel performance model for dynamic variations in processor speed, memory speed, and thread concurrency. In *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '17, pages 155–166, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4699-3. doi: 10.1145/3078597.3078601. URL http://doi.acm.org/10.1145/3078597.3078601.

[140] Cheng Li, Shihong Zou, and Lingwei Chu. Online Learning Based Internet Service Fault Diagnosis Using Active Probing. IEEE ICNSC, 2009. doi: 10.1109/ICNSC.2009.4919376.

[141] Cheng Li, I. Goiri, A. Bhattacharjee, R. Bianchini, and T.D. Nguyen. Quantifying and Improving I/O Predictability in Virtualized Systems. In *IEEE/ACM International Symposium on Quality of Service (IWQoS)*, pages 1–12, June 2013. doi: 10.1109/IWQoS.2013.6550269.

[142] Cheng Li, Philip Shilane, Fred Douglis, Darren Sawyer, and Hyong Shim. Assert(!Defined(Sequential I/O)). In *6th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 14)*, Philadelphia, PA, June 2014. USENIX Association. URL http://blogs.usenix.org/conference/hotstorage14/workshop-program/presentation/li_cheng.

[143] Cheng Li, Philip Shilane, Fred Douglis, Hyong Shim, Stephen Smaldone, and Grant Wallace. Nitro: A Capacity-Optimized SSD Cache for Primary Storage. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 501–512, Philadelphia, PA, June 2014. USENIX Association. ISBN 978-1-931971-10-2. URL http://blogs.usenix.org/conference/atc14/technical-sessions/presentation/li_cheng_1.

[144] Cheng Li, Philip Shilane, Fred Douglis, Hyong Shim, Stephen Smaldone, and Grant Wallace. Nitro: A capacity-optimized ssd cache for primary storage. In *Proceedings of the USENIX Annual Technical Conference (ATC'14)*, June 2014. ISBN 978-1-931971-10-2. URL https://www.usenix.org/conference/atc14/technical-sessions/presentation/li_cheng_1.

[145] Cheng Li, Philip Shilane, Fred Douglis, and Grant Wallace. Pannier: A Container-based Flash Cache for Compound Objects. In *Proceedings of the 16th International Middleware Conference (ACM/IFIP/USENIX Middleware 15)*, pages 50–62. ACM, 2015. ISBN 978-1-4503-3618-5. doi: 10.1145/2814576.2814734. URL http://dx.doi.org/10.1145/2814576.2814734.

[146] Cheng Li, Philip Shilane, Fred Douglis, and Grant Wallace. Pannier: A container-based flash cache for compound objects. In *Proceedings of the 16th International Middleware Conference (Middleware'15)*, December 2015.

[147] Hongxing Li, Chuan Wu, Zongpeng Li, and F.C.M. Lau. Profit-maximizing virtual machine trading in a federation of selfish clouds. In *INFOCOM, 2013 Proceedings IEEE*, pages 25–29, April 2013. doi: 10.1109/INFCOM.2013.6566728.

[148] Shen Li, Shiguang Wang, Fan Yang, Shaohan Hu, Fatemeh Saremi, and Tarek Abdelzaher. Proteus: Power proportional memory cache cluster in data centers. In *Distributed Computing Systems (ICDCS), 2013 IEEE 33rd International Conference on*, pages 73–82, 2013. doi: 10.1109/ICDCS.2013.50.

[149] Shen Li, Shaohan Hu, Shiguang Wang, Lu Su, T. Abdelzaher, I. Gupta, and R. Pace. Woha: Deadline-aware map-reduce workflow scheduling framework over hadoop clusters. In *Distributed Computing Systems (ICDCS), 2014 IEEE 34th International Conference on*, pages 93–103, June 2014. doi: 10.1109/ICDCS.2014.18.

[150] Xiaozhou Li, David G. Andersen, Michael Kaminsky, and Michael J. Freedman. Algorithmic improvements for fast concurrent cuckoo hashing. In *Proceedings of the Ninth*

European Conference on Computer Systems, EuroSys '14, pages 27:1–27:14, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2704-6. doi: 10.1145/2592798.2592820. URL http://doi.acm.org/10.1145/2592798.2592820.

[151] Zhichao Li, Amanpreet Mukker, and Erez Zadok. On the importance of evaluating storage systems' $costs. In *Proceedings of the 6th USENIX Conference on Hot Topics in Storage and File Systems*, HotStorage'14, pages 6–6, Berkeley, CA, USA, 2014. USENIX Association. URL http://dl.acm.org/citation.cfm?id=2696578.2696584.

[152] Harold Lim, Herodotos Herodotou, and Shivnath Babu. Stubby: A transformation-based optimizer for mapreduce workflows. *Proc. VLDB Endow.*, 5(11):1196–1207, July 2012. ISSN 2150-8097. doi: 10.14778/2350229.2350239. URL http://dx.doi.org/10.14778/2350229.2350239.

[153] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. Mica: A holistic approach to fast in-memory key-value storage. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 429–444, Seattle, WA, April 2014. USENIX Association. ISBN 978-1-931971-09-6. URL https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/lim.

[154] Fang Liu, Xiaokui Shu, Danfeng Yao, and Ali R. Butt. Privacy-preserving scanning of big content for sensitive data exposure with MapReduce. In *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*, CODASPY '15, pages 195–206, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3191-3. doi: 10.1145/2699026.2699106. URL http://doi.acm.org/10.1145/2699026.2699106.

[155] Fang Liu, Haipeng Cai, Gang Wang, Danfeng (Daphne) Yao, Karim O. Elish, and Barbara G. Ryder. MR-Droid: A scalable and prioritized analysis of inter-app communication risks. In *Proc. of Mobile Security Technologies (MoST)*, May 2017.

[156] Zhenhua Liu, Iris Liu, Steven Low, and Adam Wierman. Pricing data center demand response. *SIGMETRICS Perform. Eval. Rev.*, 42(1):111–123, June 2014. ISSN 0163-5999. doi: 10.1145/2637364.2592004. URL http://doi.acm.org/10.1145/2637364.2592004.

[157] Jorge Londoño, Azer Bestavros, and Shang-Hua Teng. Colocation games: And their application to distributed resource management. In *Proceedings of the 2009 Conference on Hot Topics in Cloud Computing*, HotCloud'09, Berkeley, CA, USA, 2009. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1855533.1855543.

[158] Youyou Lu, Jiwu Shu, and Weimin Zheng. Extending the lifetime of flash-based storage through reducing write amplification from file systems. In *Presented as part of the 11th USENIX Conference on File and Storage Technologies (FAST 13)*, pages 257–270, San

Jose, CA, 2013. USENIX. ISBN 978-1-931971-99-7. URL https://www.usenix.org/conference/fast13/technical-sessions/presentation/lu_youyou.

[159] Ming Mao and Marty Humphrey. Auto-scaling to minimize cost and meet application deadlines in cloud workflows. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 49:1–49:12, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0771-0. doi: 10.1145/2063384.2063449. URL http://doi.acm.org/10.1145/2063384.2063449.

[160] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache craftiness for fast multicore key-value storage. In *ACM EuroSys '12*.

[161] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 183–196, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1223-3. doi: 10.1145/2168836.2168855. URL http://doi.acm.org/10.1145/2168836.2168855.

[162] Tudor Marian, Ki Suh Lee, and Hakim Weatherspoon. Netslices: Scalable multi-core packet processing in user-space. In *Proceedings of the Eighth ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '12, pages 27–38, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1685-9. doi: 10.1145/2396556.2396563. URL http://doi.acm.org/10.1145/2396556.2396563.

[163] Leonardo Marmol, Swaminathan Sundararaman, Nisha Talagala, Raju Rangaswami, Sushma Devendrappa, Bharath Ramsundar, and Sriram Ganesan. Nvmkv: A scalable and lightweight flash aware key-value store. In *Proceedings of the 6th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage'14)*, June 2014. URL https://www.usenix.org/conference/hotstorage14/workshop-program/presentation/marmol.

[164] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. Clickos and the art of network function virtualization. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 459–473, Seattle, WA, April 2014. USENIX Association. ISBN 978-1-931971-09-6. URL https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/martins.

[165] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Syst. J.*, 9(2):78–117, June 1970. ISSN 0018-8670. doi: 10.1147/sj.92.0078. URL http://dx.doi.org/10.1147/sj.92.0078.

[166] Lyle A. McGeoch and Daniel D. Sleator. A strongly competitive randomized paging algorithm. *Algorithmica*, 6(1-6):816–825, 1991. ISSN 0178-4617. doi: 10.1007/BF01759073. URL http://dx.doi.org/10.1007/BF01759073.

[167] Fei Meng, Li Zhou, Xiaosong Ma, Sandeep Uttamchandani, and Deng Liu. vcacheshare: Automated server flash cache space management in a virtualization environment. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 133–144, Philadelphia, PA, June 2014. USENIX Association. ISBN 978-1-931971-10-2. URL https://www.usenix.org/conference/atc14/technical-sessions/presentation/meng.

[168] Micron. Micron MLC SSD Specification. http://www.micron.com/products/nand-flash, 2013.

[169] Madalin Mihailescu, Gokul Soundararajan, and Cristiana Amza. Mixapart: Decoupled analytics for shared storage systems. In *Presented as part of the 11th USENIX Conference on File and Storage Technologies (FAST 13)*, pages 133–146, San Jose, CA, 2013. USENIX. ISBN 978-1-931971-99-7. URL https://www.usenix.org/conference/fast13/technical-sessions/presentation/mihailescu.

[170] Dushyanth Narayanan, Austin Donnelly, and Antony Rowstron. Write off-loading: Practical power management for enterprise storage. In *Proceedings of 6th USENIX Conference on File and Storage Technologies (FAST'08)*, February 2008. URL http://research.microsoft.com/apps/pubs/default.aspx?id=68027.

[171] Dushyanth Narayanan, Eno Thereska, Austin Donnelly, Sameh Elnikety, and Antony Rowstron. Migrating server storage to ssds: Analysis of tradeoffs. In *Proceedings of the 4th ACM European Conference on Computer Systems*, EuroSys '09, pages 145–158, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-482-9. doi: 10.1145/1519065.1519081. URL http://doi.acm.org/10.1145/1519065.1519081.

[172] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling memcache at facebook. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 385–398, Lombard, IL, 2013. USENIX. ISBN 978-1-931971-00-3. URL https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/nishtala.

[173] Yongseok Oh, Jongmoo Choi, Donghee Lee, and Sam H. Noh. Caching less for better performance: Balancing cache size and update cost of flash memory cache in hybrid storage systems. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST'12)*, February 2012. URL http://dl.acm.org/citation.cfm?id=2208461.2208486.

[174] Yongseok Oh, Eunjae Lee, Choulseung Hyun, Jongmoo Choi, Donghee Lee, and Sam H. Noh. Enabling cost-effective flash based caching with an array of commodity ssds. In *Proceedings of the 16th Annual Middleware Conference (Middleware'15)*, December 2015.

[175] Diego Ongaro, Stephen M. Rumble, Ryan Stutsman, John Ousterhout, and Mendel Rosenblum. Fast crash recovery in ramcloud. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 29–41, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0977-6. doi: 10.1145/2043556.2043560. URL http://doi.acm.org/10.1145/2043556.2043560.

[176] Jian Ouyang, Shiding Lin, Song Jiang, Zhenyu Hou, Yong Wang, and Yuanzheng Wang. Sdf: Software-defined flash for web-scale internet storage systems. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'14)*. ACM, 2014. ISBN 978-1-4503-2305-5. doi: 10.1145/2541940.2541959. URL http://doi.acm.org/10.1145/2541940.2541959.

[177] Amar Phanishayee, Elie Krevat, Vijay Vasudevan, David G. Andersen, Gregory R. Ganger, Garth A. Gibson, and Srinivasan Seshan. Measurement and analysis of tcp throughput collapse in cluster-based storage systems. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, FAST'08, pages 12:1–12:14, Berkeley, CA, USA, 2008. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1364813.1364825.

[178] Krishna P.N. Puttaswamy, Thyaga Nandagopal, and Murali Kodialam. Frugal storage for cloud file systems. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 71–84, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1223-3. doi: 10.1145/2168836.2168845. URL http://doi.acm.org/10.1145/2168836.2168845.

[179] Dai Qin, Angela Demke Brown, and Ashvin Goel. Reliable writeback for client-side flash caches. In *Proceedings of the USENIX Annual Technical Conference (ATC'14)*, June 2014. ISBN 978-1-931971-10-2. URL https://www.usenix.org/conference/atc14/technical-sessions/presentation/qin.

[180] Ananth Rao, Karthik Lakshminarayanan, Sonesh Surana, Rickard Karp, and Ion Stoica. Load balancing in structured p2p systems. In *IPTPS'03*, 2003.

[181] Ricardo Santana, Steven Lyons, Ricardo Koller, Raju Rangaswami, and Jason Liu. To ARC or not to ARC. In *Proceedings of the 7th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage'15)*, July 2015. URL https://www.usenix.org/conference/hotstorage15/workshop-program/presentation/santana.

[182] Mohit Saxena, Michael M. Swift, and Yiying Zhang. Flashtier: A lightweight, consistent and durable storage cache. In *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys'12)*, 2012. ISBN 978-1-4503-1223-3. doi: 10.1145/2168836.2168863. URL http://doi.acm.org/10.1145/2168836.2168863.

[183] Weijie Shi, Linquan Zhang, Chuan Wu, Zongpeng Li, and Francis C.M. Lau. An online auction framework for dynamic resource provisioning in cloud computing. In *The 2014 ACM International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '14, pages 71–83, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2789-3. doi: 10.1145/2591971.2591980. URL http://doi.acm.org/10.1145/2591971.2591980.

[184] Hyong Shim, Philip Shilane, and Windsor Hsu. Characterization of incremental data changes for efficient data protection. In *Proceedings of the USENIX Annual Technical Conference (ATC'13)*, 2013. ISBN 978-1-931971-01-0. URL https://www.usenix.org/conference/atc13/technical-sessions/presentation/shim.

[185] Xiaokui Shu, Fang Liu, and Danfeng (Daphne) Yao. *Rapid Screening of Big Data Against Inadvertent Leaks*, pages 193–235. Springer International Publishing, Cham, 2016. ISBN 978-3-319-27763-9. doi: 10.1007/978-3-319-27763-9_5. URL http://dx.doi.org/10.1007/978-3-319-27763-9_5.

[186] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-1-4244-7152-2. doi: 10.1109/MSST.2010.5496972. URL http://dx.doi.org/10.1109/MSST.2010.5496972.

[187] Roshan Sumbaly, Jay Kreps, Lei Gao, Alex Feinberg, Chinmay Soman, and Sam Shah. Serving large-scale batch computed data with project voldemort. In *USENIX FAST*, 2012.

[188] Linpeng Tang, Qi Huang, Wyatt Lloyd, Sanjeev Kumar, and Kai Li. Ripq: Advanced photo caching on flash for facebook. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST'15)*, February 2015. ISBN 978-1-931971-201. URL https://www.usenix.org/conference/fast15/technical-sessions/presentation/tang.

[189] Olivier Temam. Investigating optimal local memory performance. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*. ACM, 1998. ISBN 1-58113-107-0. doi: 10.1145/291069.291050. URL http://doi.acm.org/10.1145/291069.291050.

[190] TRIM. TRIM Specification. ATA/ATAPI Command Set- 2 (ACS-2). http://www.t13.org/.

[191] Robbert van Renesse and Fred B. Schneider. Chain replication for supporting high throughput and availability. In *USENIX OSDI'04*.

[192] Abhishek Verma, Ludmila Cherkasova, and Roy H. Campbell. Play it again, simmr! In *Proceedings of the 2011 IEEE International Conference on Cluster Computing*, CLUSTER '11, pages 253–261, Washington, DC, USA, 2011. IEEE Computer Society. ISBN 978-0-7695-4516-5. doi: 10.1109/CLUSTER.2011.36. URL http://dx.doi.org/10.1109/CLUSTER.2011.36.

[193] Guanying Wang, A.R. Butt, P. Pandey, and K. Gupta. A simulation approach to evaluating design decisions in mapreduce setups. In *Modeling, Analysis Simulation of Computer and Telecommunication Systems, 2009. MASCOTS '09. IEEE International Symposium on*, pages 1–11, Sept 2009. doi: 10.1109/MASCOT.2009.5366973.

[194] Hao Wang, Weifeng Liu, Kaixi Hou, and Wu-chun Feng. Parallel Transposition of Sparse Data Structures. In *Proceedings of the 2016 International Conference on Supercomputing*, ICS '16, pages 33:1–33:13, New York, NY, USA, 2016. ACM.

[195] Hui Wang and Peter Varman. Balancing fairness and efficiency in tiered storage systems with bottleneck-aware allocation. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST 14)*, pages 229–242, Santa Clara, CA, 2014. USENIX. ISBN ISBN 978-1-931971-08-9. URL https://www.usenix.org/conference/fast14/technical-sessions/presentation/wang.

[196] Alexander Wieder, Parmod Bhatotia, Ansley Post, and Rodrigo Rodrigues. Orchestrating the deployment of computations in the cloud with conductor. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 367–381, San Jose, CA, 2012. USENIX. ISBN 978-931971-92-8. URL https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/wieder.

[197] Alex Wiggins and Jimmy Langston. Enhancing the scalability of memcached, 2012.

[198] Guanying Wu and Xubin He. Delta-ftl: Improving ssd lifetime via exploiting content locality. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 253–266, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1223-3. doi: 10.1145/2168836.2168862. URL http://doi.acm.org/10.1145/2168836.2168862.

[199] Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Characterizing facebook's memcached workload. *Internet Computing, IEEE*, 18(2):41–49, Mar 2014. ISSN 1089-7801. doi: 10.1109/MIC.2013.80.

[200] Gala Yadgar, Michael Factor, and Assaf Schuster. Karma: Know-it-all replacement for a multilevel cache. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST'07)*, February 2007. URL http://dl.acm.org/citation.cfm?id=1267903.1267928.

[201] Yaffs. Yaffs (Yet Another Flash File System). http://www.yaffs.net/.

[202] Jingpei Yang, Ned Plasson, Greg Gillis, and Nisha Talagala. Hec: Improving endurance of high performance flash-based cache devices. In *Proceedings of the 6th International Systems and Storage Conference (SYSTOR'13)*. ACM, 2013. ISBN 978-1-4503-2116-7. doi: 10.1145/2485732.2485743. URL http://doi.acm.org/10.1145/2485732.2485743.

[203] Dong Yuan, Yun Yang, Xiao Liu, and Jinjun Chen. A data placement strategy in scientific cloud workflows. *Future Generation Computer Systems*, 26(8):1200 – 1214, 2010. ISSN 0167-739X. doi: http://dx.doi.org/10.1016/j.future.2010.02.004. URL http://www.sciencedirect.com/science/article/pii/S0167739X10000208.

[204] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of USENIX NSDI 2012*. ISBN 978-931971-92-8. URL https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia.

[205] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 265–278, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-577-2. doi: 10.1145/1755913.1755940. URL http://doi.acm.org/10.1145/1755913.1755940.

[206] Da Zhang, Hao Wang, Kaixi Hou, Jing Zhang, and Wu chun Feng. pDindel: Accelerating InDel Detection on a Multicore CPU Architecture with SIMD. In *2015 IEEE 5th International Conference on Computational Advances in Bio and Medical Sciences (ICCABS)*, pages 1–6, Oct 2015.

[207] Wei Zhang, Jinho Hwang, Timothy Wood, K.K. Ramakrishnan, and Howie Huang. Load balancing of heterogeneous workloads in memcached clusters. In *9th International Workshop on Feedback Computing (Feedback Computing 14)*, Philadelphia, PA, June 2014. USENIX Association. URL https://www.usenix.org/conference/feedbackcomputing14/workshop-program/presentation/zhang.

[208] Yifeng Zhu and Hong Jiang. Race: A robust adaptive caching strategy for buffer cache. *IEEE Trans. Comput.*, 57(1):25–40, January 2008. ISSN 0018-9340. doi: 10.1109/TC.2007.70788. URL http://dx.doi.org/10.1109/TC.2007.70788.