Towards Efficient and Flexible Object Storage Using Resource and Functional Partitioning

Ali Anwar

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
Dennis G. Kafura
Dimitrios S. Nikolopoulos
Eli Tilevich
Daphne Yao

May 03, 2018
Blacksburg, Virginia

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

Towards Efficient and Flexible Object Storage Using Resource and Functional Partitioning

Ali Anwar

(ABSTRACT)

The big data boom is driving the development of innovative distributed storage systems (e.g., key-value stores, cloud storage services, distributed filesystems, and flash cache, etc.) aimed at meeting the increasing need for storing vast volumes of data. 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 user requirements. 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 implement a series of novel techniques, algorithms, and frameworks, to realize efficient and flexible object stores. We select three different application scenarios—cloud object stores, distributed key-value storage systems, and storage for containers—targeting two general modern workloads—internet-scale web workloads and IBM Docker containers workloads. The overarching goal of this dissertation is to improve the efficiency and flexibility of modern storage applications by using resource and functional partitioning.

In the first part of this dissertation (Chapter 3), we focus on how the monitoring data is collected by Cloud providers and present solutions for the open problems. In the second part of this dissertation (Chapter 4), we design MOS, an object store that (i) dynamically provisions microstores, each configured with different combination of hardware and software options, and (ii) exposes the interfaces of microstores to the tenants to use according to application requirements. Furthermore, we extend our basic framework to MOS$_{++}$ that uses container based approach to launch resources in a more fine-grained manner. In the third part of this dissertation (Chapter 5), 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. In the fourth part of this dissertation (Chapter 6), for the first time in the known literature, we perform a large-scale and comprehensive analysis of a real-world Docker registry workload. Based on our findings, we derive several design implications for container registry services.

My dissertation shows that by intelligently employing resource and functional partitioning strategies, efficiency (i.e., performance, monetary cost, etc.) and flexibility (i.e., ease-of-use, ease-of-deployment, programmability, etc.) of object stores can be improved by manyfolds. The principles of leveraging modularity can be used to guide next-generation storage system software design, especially when being faced with new storage hardware technologies.

Towards Efficient and Flexible Object Storage Using Resource and Functional Partitioning

Ali Anwar

(GENERAL AUDIENCE ABSTRACT)

Modern storage systems are designed to manage data without considering the dynamicity of user or resource requirements. This design approach does not consider the complexities of the dynamically changing runtime application behaviors as well as the unique features of underlying resources. To this end, this dissertation studies how resource and functional partitioning strategies can improve efficiency and flexibility of object stores. This dissertation presents a series of practical and efficient techniques, algorithms, and optimizations to realize efficient and flexible object stores. The experimental evaluation demonstrates the effectiveness of our design choices and strategies to make object stores flexible and resource-aware.

Dedicated to my parents Nighat Yasmeen and Muhammad Anwar, and my wife Muniba Fayyaz, for their support, motivation, encouragement, and love.

Acknowledgments

This Ph.D. would not have been possible without the support and encouragement from many people; I would like to express my deepest gratitude to the following awesome individuals!

First, I would like to thank my advisor Ali R. Butt for his constant support and guidance. Ali encouraged me to work at DSSL and gave me sufficient space to grow from a graduate student to an independent researcher. Ali has taught me everything that was needed to succeed in graduate school life such as, how to read/write a research paper, what are necessary skills to deliver good presentation, how to communicate with peers and establish a good and productive collaboration relationship, etc.. He is supportive and encouraging on everything I did, both academically and otherwise. Without Ali, I would have never achieved this goal.

I would also like to thank my Ph.D. committee members: Professors Dennis Kafura, Dimitrios S. Nikolopoulos, Eli Tilevich, and Daphne Yao, for their insightful comments and feedback on my dissertation at various stages of my Ph.D.

I have been fortunate to work with many other researchers and faculty members at Virginia Tech and other institutes. I am grateful to Anca Sailer, Andrzej Kochut, Charles O. Schulz, Alla Segal (IBM Research T.J. Watson) for their valueable insights for Monitoring and Metering project. I am grateful to Aayush Gupta (IBM Research Almaden) for his guidance on MOS project. I am grateful to Hai Huang (IBM Research T.J. Watson), Dongyoon Lee Tilevich, Godmar Back, Fred Douglis (Vencore Labs) for their guidance on ClusterOn project. I am gratefule to Mohamed Mohamed, Vasily Tarasov, and Lukas Rupprecht (IBM Research Almaden) for their guidance for Docker Registry project. I am grateful to Michael Littley for his help on Docker registry project. Last but not least, I am really grateful to Yue Cheng for his help in MOS and ClusterOn project.

My work as a Ph.D. student was made possible, and constantly enjoyable, due to collaboration with my lab-mate, and one of my closest friends, Yue Cheng. Yue has been constant source of inspiration for me through out my stay in VT. His passion for research, work ethics, friendly nature, and willingness to discuss and share ideas were the reason we co-authored 6 papers and hopefully more are coming. During my stay at DSSL, most of my time was spent with Yue brainstorming different projects and arguing about novelty and contribution

of each work we did together. We spent several sleepless nights hacking different systems, writing code for ClusterOn project, running experiments, and arguing over results. I will never forget those lengthy discussions, not just about research but also about stuff going around in our life. Yue has been a sounding board for me through my stay in VT. Without Yue, grad school life would have been very boring.

I made several friends during my stay at Blacksburg and Safdar Iqbal is one the closest friend among those. I met Safdar before coming to Virginia Tech at an orientation for graduate students. Our common interest in TV shows and movies made our friendship even stronger. I will always cherish the good memories of Kroger trips and cooking sessions. I will always be grateful to Safdar for his support and encouragement. Safdar Iqbal helped a great deal during my early days of PhD. He taught me difference between good and bad research and for this I will always be grateful to him. Best of luck on your new role in industry.

I am extremely fortunate to have worked with a wonderful group of lab-mates at DSSL. A shout-out to my DSSL labmates at Virginia Tech: Min, Alex, Hafeez, Krish, Hyogi, Luna, Nannan, Arnab, Bharti, Salman, Sangeetha, Arjun, Jingoo, Jamal, and Michael. Our social interactions have enriched my grad school life.

In closing, I am immensely grateful to my family, without whom this Ph.D. would have been lightweight and meaningless. My dad, Muhammad Anwar, always encouraged and supported me. My mom, Nighat Yasmeen, 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 grandparents for their love throughout my life. I would like to thank my cousins Khawar, Faiza and Saima for their encouragement and support ever since I was in middle school. I would like to thank my siblings, Faria, Fatima, and Talha, for providing constant support and help. My wife, Muniba, 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 and friends.

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:

- Anca Sailer, Andrzej Kochut, Charles O. Schulz, Alla Segal contributed to the work included in Chapter 3 of the dissertation which resulted in [69, 72, 75].
- Aayush Gupta contributed to the resource partitioning of the object storage [71, 76] in Chapter 4.

- Yue Cheng contributed to this dissertation in multiple joint papers [67, 68, 71, 76, 77]. He made a great amount of contribution in both design and evaluation of the work in Chapter 5.
- Hai Huang, Dongyoon Lee, and Fred Douglis contributed to the work in Chapter 5. Especially, Hai contributed a significant amount to the overall idea and system design of ClusterOn.
- Mohamed Mohamed, Vasily Tarasov, Michael Littley, Lukas Rupprecht, Dimitrios Skourtis, Amit S. Warke, Heiko Ludwig, Dean Hildebran contributed to [77] in Chapter 6.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Distributed object stores such as in-memorry key-value (KV) stores and persistent cloud object stores play an increasingly critical role in supporting today's large-scale applications. Relatively simple data schemas and indexing enable KV stores to achieve high performance and scalability, as well as quick request processing. Amazon employs Dynamo [105] for online shopping service with eventual consistency guarantee; Expedia is powered by the strongly consistent MongoDB [40] for serving the flight and trip reservation services. Similarly, cloud object stores, such as Amazon S3, Google Cloud Store (GCS), OpenStack Swift and Ceph [166], have become the most widely used form of persistent cloud storage in recent years. These stores combine key advantages such as high availability, elasticity and a "pay-as-you-go" pricing model, which allows applications to scale as the usage increases or decreases, and offers HTTP-based RESTful APIs for easy data management. The desirable features, coupled with the advances in virtualization infrastructure, are driving the adoption of distributed object stores by a myriad of applications. Examples range from web applications [80] that store image and video files, to backup services [28] that require large capacity for archival data, to big data analytics frameworks [95].

1.1 Motivation

A typical deployment of cloud object stores either opts to use a monolithic configuration or segmented storage setup [57] with a static configuration to handle different types of applications with evolving requirements. From the cloud provider's perspective, supporting dramatically different workloads from different applications (tenants) using a single homogeneous configuration means that optimization opportunities are lost. The situation is further complicated by the fact that due to regular system upgrades and introduction of new storage architectures, data centers hosting the object stores are becoming increasingly heterogeneous [107, 145].

The object storage system 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 object storage system from scratch imposes non-trivial engineering efforts in terms of # lines of code (LoC) or person-year. 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"*.

To address the above issues, this dissertation proposes, designs, and implements a series of novel techniques, algorithms, and frameworks, to efficient and flexible object stores. This dissertation selects three different application scenarios—cloud object stores, distributed key-value storage systems, and storage for containers—targeting two general modern workloads—internet-scale web workloads and IBM Docker containers workloads. The overarching goal of this dissertation is to improve the efficiency and flexibility of modern storage applications by using resource and functional partitioning.

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

1.1.1 Monitoring and Metering Cloud Solutions

The cloud computing model has emerged as the de facto paradigm for efficiently providing infrastructure, platform, and application services for IT industry. As a result, vendors such as IBM, Amazon, and RedHat offer cloud based solutions to optimize the use of their data centers. Customers of these cloud providers, particularly those building their critical production businesses on cloud services, are interested in detailed monitoring data to track in real time the health of their thousands of service instances. Netflix, for instance, collects tens of thousands of metrics per microservice every 1 to 5 seconds [1]. With service management datasets growing at a rate of 10 Billion+ records/day, typical monitoring and analysis tools break or require significant additional capacity for storage and computation.

Hence there is a clear need to understand and improve the design of underlying architecture provided by open source cloud solutions, such as OpenStack, to efficiently carry out monitoring and metering tasks. In this work, we focus on how the monitoring data is collected by Cloud providers and present solutions for the open problems.

1.1.2 Resource Partitioning of Object Stores

A typical deployment of cloud object stores either opts to use a monolithic configuration or segmented storage setup [57] with a static configuration to handle different types of applications with evolving requirements. Using a monolithic configuration setup results in all applications experiencing the same service level, e.g., similar average latency per request, data transfer throughput, and queries per second (QPS). However, different applications entail extremely different latency and throughput requirements. For example, a social networking or photo sharing application requires low latency to support a highly-responsive user experience, whereas backup services can tolerate higher latency but require sustained high throughput.

Hence in this work, we posit that compared to using a rigid object store *it is more beneficial to support multi-tenant workloads separately using dynamically configurable finer-grained object stores* on sub-clusters of available resources.

1.1.3 Functional Partitioning of Object 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.

This part of the 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.

1.1.4 Efficient Object Storage for Container Distribution

To facilitate the deployment of microservices, Docker provides a *registry service*. The registry acts as a central image repository that allows users to publish their images and make them

accessible to others. To run a specific software component, users then only need to "pull" the required image from the registry into local storage. A variety of Docker registry deployments exist such as Docker Hub [13], IBM Bluemix Container Registry [31], or Artifactory [4].

Several recent studies have proposed novel approaches to improve Docker client and registry communication [117, 126, 150]. However, these studies suffer from a serious shortcoming—they used unreasonably small, oversimplified, or synthetic Docker workloads. In this work, for the first time in the known literature, we perform a large-scale and comprehensive analysis of real-world Docker registry workload.

1.2 Research Contributions

From the above aspects, *we demonstrate in this dissertation that we can improve the storage performance and cost efficiency by adding resource and functional partitioning.*

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

Monitoring and Metering Cloud Solutions In this work, we first present a deep analysis of how open source cloud solutions, such as OpenStack, facilitates monitoring and metering. Second, by specifically focusing on how the monitoring data is collected by OpenStack and by analyzing the collected data we identify open problems in OpenStack. Third, present solutions for those open problems and discuss the challenges therein. We believe that we have looked deeply into how monitoring subsystem provided by open source cloud solutions collect data.

Resource Partitioning of Object Stores In this work, we first evaluate the impact of conventional object storage configuration on performance and resource efficiency by conducting experiments on a local Swift testbed. Our observations stress the need to carefully evaluate the various configuration choices and develop simple *Rules-of-Thumb* that cloud providers can leverage for provisioning the object storage. Second, we perform a detailed performance and resource efficiency analysis on identifying major hardware and software configuration opportunities that can be used to fine-tune object stores for specific workloads. Our findings indicate the need to re-architect cloud object storage specialized for the public cloud. Third, based on our behavior analysis, we design MOS, an object store that (i) dynamically provisions microstores, each configured with different combination of hardware and software options, and (ii) exposes the interfaces of microstores to the tenants to use according to application requirements. Fourth, we extend our basic framework to MOS$_{++}$ that uses container based approach to launch resources in a more fine-grained manner. MOS$_{++}$ is SLA-aware, supports rapid deployment, portability across machines, offers a lightweight footprint,

and simplifies maintenance. Fifth, we implement a prototype of MOS$_{++}$, and demonstrate that our approach results in improved performance (by up to 89.6% and 79.8% compared to the default monolithic and statically configured object store setup, respectively), as well as higher resource efficiency. Furthermore, we design a simulator to evaluate our solution under a large-scale 456-core cloud cluster setup. We also compare the performance of MOS with MOS$_{++}$ to highlight advantages of our container based approach.

Functional Partitioning of Object 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 [43], SSDB [55], and Masstree [144]. 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.

Efficient Object Storage for Container Distribution In this work, we first collect and anonymize large-scale production-level traces for IBM Bluemix container registry that supports a breadth of representative container applications and uses. Second, we design and implement a trace replayer for Docker which can be used to evaluate the existing registry deployment under different configurations. The traces and the replayer are publicly available at `https://dssl.cs.vt.edu/drtp/`. Third, using the traces and the replayer, we perform several comprehensive analysis of Docker registry workload and infer a number of crucial insights about container workloads. Fourth, based on our analysis, we propose optimizations that reduce request latencies for the Docker registry.

1.3 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 solution of effective monitoring and meetering of cloud resources. Chapter 4 presents workload-aware elasticity for cloud object stores. Chapter 5 introduces a flexible framework that helps reduce the required engineering

effort when building a new distributed key-value storage system from scratch. Chapter 6 describes in-depth analysis of a large scale Docker registry to make conscious decisions when designing storage for containers and Docker in particular. Chapter 7 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 functional and resource partitioning to object storage solutions. 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 techniques and algorithms in this dissertation.

2.1 Monitoring and Metering of Cloud Solutions

Scalable Cloud Monitoring The focus of several recent works [73, 74, 82, 125, 154] is on providing an efficient and scalable cloud monitoring setup, however, these works do not consider or discuss reduction in collected monitoring data. Similarly, some other works in distributed state monitoring such as [127, 148] are either to study the problem of employing distributed constraints to minimize the communication cost or to ensure trustworthiness of resource accounting [88, 142]. While these works either provide communication efficient detection or verifiable resource accounting, we study the lower level problem on efficiently collecting monitoring data for both anomaly detection as well as cloud metering purposes. Recently, there has also been work done to evaluate the idea of using TimeSeries storage for the metering data but unlike us the goal of this work is limited to reducing the avgObjSize of each sample [11, 50].

Anomaly Detection Volley [149] proposes violation likelihood based state monitoring for datacenters and perhaps is closest to our work as it also utilizes node-level adaption algorithms to minimize monitoring cost. Similalrly,[93, 94, 96, 98, 100] do monitoring from pricing perspective. Our work differs from Volley in a sense that we not only study the effect of controlled monitoring on anomaly detection but also study its impact on cloud metering. Finally, some works make assumption on value distribution[85] in clustering VMs for multi-cloud systems, while our approach makes no such assumption.

2.2 Resource Partitioning of Object Stores

Object Store Segmentation. Swift provides storage policies to support for segmenting the cluster through the creation of multiple object rings [57]. This feature is useful if a provider wants to offer different level of durability, performance, or storage implementation but does not want to maintain separate clusters. In contrast, MOS advocates to separately maintain clusters to incorporate segmentation across all layers. Furthermore, storage policies are static whereas MOS dynamically perform resource provisioning that can drive online reconfiguration across multiple partitions of the object store. Also, performance comparison of the Swift-based prototype of MOS with other object stores like Ceph will not be an apple-to-apple comparison as Ceph outperforms Swift [6] and they have significantly different architecture.

Workload-aware Elasticity. The focus of various recent research works have been on providing an elastic setup for cloud based storage [84, 92, 99]. Lim et al. [139] propose an elastic storage system on HDFS for multi-tier application services. Similarly, ElasTraS [104] provides scalability and elasticity to the data store in clouds for optimizing transactional data access. MeT [103] focuses on systems metrics (CPU utilization, I/O wait and memory usage) that are critical for a NoSQL database. Skute [81] provides a fault-tolerance and scalable replication scheme for cloud storage. MOS differs from these works in that it focuses on providing best performance guarantee for heterogeneous multi-tenant workloads by exploiting automated elasticity for cloud object store.

Handling Cluster/Workload Heterogeneity. hatS [129] proposes a replication scheme for HDFS that integrates heterogeneous storage technologies into Hadoop. ϕSched [128] designs a cluster-heterogeneity-aware scheduler to improve the resource-application match. Walnut [89] suggests using a hybrid object strategy to support both small and large objects in an object store. CAST [95] and its extension [97] perform coarse-grained cloud storage (including object stores) management for data analytics workloads. In contrast, MOS explicitly partitions the conventional monolithic storage into multiple dynamically tuned microstores, each serving a particular type of workload.

Meeting SLA/SLO. SCADS [163] uses a steady-state performance model to predict whether a server can handle a particular workload, without violating a given latency threshold. SCADS reconfigures the storage system on-the-fly in response to workload changes driven by a performance model. Similarly, Papio [159] introduces a QoS-enabled function into the S3-based object store where it accepts an explicit performance request as an advanced reservation, and enables QoS in the access with the extended S3 RESTful interfaces. MOS differs from these works in that it: i) keeps track of fine-grained resource usage; and ii) partitions the heterogeneous workloads and optimizes each individually based on tenants' SLA requirements while yielding higher overall performance and better resource efficiency.

Dynamic Resource Management. Mantle [155] is a programmable storage system that

lets users inject custom balancing logic into Ceph [166]. This feature provides flexibility for allocating resources. Unlike Mantle, MOS automatically performs resource provisioning without burgeoning the users. [78] propose a fine-grained resource allocation mechanism based on metrics such as CPU utilization, I/O wait and memory usage that are critical for MapReduce workloads. Similarly, Lee et al. [133] design a heterogeneity-aware resource allocator and job scheduler for a cloud data analytics system. While these works provide heterogeneity-aware optimizations targeted at MapReduce workloads, MOS targets object stores and focuses on improving the performance of real-time request processing.

Software-Defined Storage. IOFlow [162] provides a software-defined storage architecture by enforcing policies to dictate a storage I/O flow's performance and routing. IOFlow provides ease of deployment by providing a queue and control abstraction at two OS stages — the storage drivers in the hypervisor and the storage server. However, MOS serves as a transparent middleware layer that provides dynamic resource provisioning beneath any object stores. More importantly, MOS requires no OS-level modifications.

2.3 Functional Partitioning of Object Stores

Distributed management for storage engine Dynomite [19] and ElastiCache [20] adds fault tolerance and consistency support for simple data stores such as Redis. Dynomite only supports eventual consistency with AA topology. It also requires the single-server applications to support distributed management functions such as Redis's streaming data recovery/migration mechanism. Dynomite requires the developers to fully understand its internal designs while developers of ClusterOn datalets are oblivious of these details. ClusterOn's datalet is completely oblivious of the upper-level distributed management, which offers improved flexibility and programmability via control and data plane isolation.

Tuneable consistency Pileus [161] is a cloud storage system that offers a range of consistency-level SLAs. Some storage systems offer tunable consistency, e.g., ManhattanDB [36]. FlexKV [153] is another flexible key-value store that can be configured to act as a non-persistent/durable store and operates consistently/inconsistently. Morphus [113] provides support towards reconfigurations for NoSQL stores in an online manner. To the best of our knowledge, ClusterOn is the first generic framework that offers a broad range of consistency/topology options for both users and KV store application developers.

Proxy-based KV stores A large body of research has focused on improving the performance of proxy-based layered distributed KV stores [110, 137, 169, 170]. Although these works propose a general method to improve distributed application performance and efficiency, However, unlike ClusterOn, such works do not provide a generic pluggable framework for supporting a variety of backends.

Binary-level partitioning Coign [120] automatically converts a program into a distributed

model by performing static code analysis and binary-level partitioning. Similar to ClusterOn, Coign allows end users (without access to source code) to transform a non-distributed application into a distributed application by leveraging a graph model and a graph-cutting algorithm for application partitioning. In contrast, ClusterOn is nonintrusive and more scalable, as it seamlessly scales out a non-distributed application instead of partitioning it.

2.4 Storage for Container Distribution

Docker containers Improving performance of container storage has recently attracted attention from both industry and academia. DRR [108] improves common copy-on-write performance targeting a dense container-intensive workload. Tarasov et al. [160] study the impact of the storage driver choice on the performance of Docker containers for different workloads running inside the containers. Contrary to this work, we focus on the registry side of a container workload.

Docker registry Other works have looked at optimizing image retrieval from a registry side [117, 150]. Slacker [117] speeds up the container startup time by utilizing lazy cloning and lazy propagation. Images are fetched from a shared NFS store and only the minimal amount of data needed to start the container is retrieved initially. Additional data is fetched on demand. However, this design tightens the integration between the registry and the Docker client as clients now need to be connected to the registry at all times (via NFS) in case additional image data is required. Contrariwise, our study focuses on the current state-of-the-art Docker deployment in which the registry is an independent instance and completely decoupled from the clients.

CoMICon [150] proposes a system for cooperative management of Docker images among a set of nodes using peer-to-peer (P2P) protocol. In its essence, CoMICon attempts to fetch a missing layer from a node in close proximity before asking a remote registry for it. Our work is orthogonal to this approach as it analyzes a registry production workload. The results of our analysis and the collected traces can also be used to evaluate new registry designs such as CoMICon.

To the best of our knowledge, similar to IBM Cloud, most public registries [13, 23, 25] use the open-source implementation of the Docker registry [15]. Our findings are applicable to all such registry deployments.

Workload analysis studies A number of works [79, 119] have studied web service workloads to better understand how complex distributed systems behave at scale. Similar studies exist [90, 91] which focus on storage and file system workloads to understand access patterns and locate performance bottlenecks. No prior work has explored the emerging container workloads in depth.

Slacker [117] also includes the HelloBench [27] benchmark to analyze push/pull performance

of images. However, Slacker looks at client-side performance while our analysis is focused at registry side. Our work takes a first step in performing a comprehensive and large-scale study on real-world Docker container registries.

Caching and prefetching Caching and prefetching have long been effective techniques to improve system performance. For example, modern datacenters use distributed memory cache servers [37, 47, 93, 94] to improve database query performance by caching the query results. Moreover, Redis [47], which is a more versatile in-memory key-value store, serves similar function in modern datacenters. More work targeting caching and prefetching to optimize systems performance such as [167] and [138] has been conducted. A large body of research [83, 86, 114, 136, 152, 167, 171] studied the effects of combining caching and prefetching. In our work we demonstrate that the addition of caches significantly improves container registry's performance, while layer prefetching reduces the pull latency for large and less popular images.

Chapter 3

Monitoring and Metering of Cloud Solutions

3.1 Introduction

The cloud computing model has emerged as the de facto paradigm for efficiently providing infrastructure, platform, and application services for IT industry. As a result, vendors such as IBM, Amazon, and RedHat offer cloud based solutions to optimize the use of their data centers. Customers of these cloud providers, particularly those building their critical production businesses on cloud services, are interested in detailed monitoring data to track in real time the health of their thousands of service instances. Netflix, for instance, collects tens of thousands of metrics per microservice every 1 to 5 seconds [1]. With service management datasets growing at a rate of 10 Billion+ records/day, typical monitoring and analysis tools break or require significant additional capacity for storage and computation.

Furthermore, the charge model most sought after by the customers of those providers is the fine-grained pay-per-use, where users are charged for the amount of specific resources, e.g., volume of transactions, CPU usage, etc., consumed during a given time period [124]. The cloud service providers, looking to maintain the competitive advantage by effectively adapting to versatile charging policies, have started to promote pay-per-use. However, usage based pricing brings a new set of service management requirements for the service providers, particularly for their revenue management [164]. The finer-grain metering requires monitoring of service resources and applications at appropriate level to provide useful information about the resource consumption that is to be charged for. This may result in collecting significantly large amounts of metered data. Additionally, this metered data needs computational resources to be processed in order to perform revenue management specific tasks.

The resource capacity requirement for such non-revenue generating systems such as monitoring and metering fluctuates largely with the service demand (e.g., the number of service

instances), the service price policy updates (e.g., from single metric based charge to complex multi-metric based charge), the resolution of the system behavior exposed (e.g., from higher-level aggregations to individual runaway thread), while their unit cost changes depending on the operational infrastructure solution (e.g., on premise, traditional outsourcing or IaaS). A crucial challenge for the cloud service providers and their customers is how to control the quickly escalating data size of their service management, and implicitly its costs, in order to profitably remain in the race for the cloud market.

Hence there is a clear need to understand and improve the design of underlying architecture provided by open source cloud solutions, such as OpenStack, to efficiently carry out monitoring and metering tasks. In this work, we present a deep analysis of how OpenStack facilitates monitoring and metering. We specifically focus on how the monitoring data is collected by OpenStack and by analyzing the collected data we identify open problems in OpenStack. Furthermore, we also present solutions for those open problems and discuss the challenges therein.

3.2 OpenStack – Ceilometer Background

In OpenStack, Ceilometer [46] is the module that provides an infrastructure to collect detailed measurements about resources managed by OpenStack. The main components of ceilometer can be divided into two categories, namely agents, e.g., compute agents, central agents, etc., and services, e.g., collector service, API service, etc. The compute agents poll the local `libvirt` daemon to fetch resource utilization of launched VMs and emit this data as AMQP [2] notifications on the message bus called Ceilometer bus. Similarly, central agents poll the public RESTful APIs of OpenStack services, such as Cinder and Glance, to track resources and emit this data onto the OpenStack's common message bus called Notification bus. On the other hand, the collector service collects the AMQP notifications from the agents and other OpenStack services, and dispatches the collected information to the metering database. Finally, the job of the API service is to present aggregated metering data to the billing engine. In Ceilometer, resource usage measurement, e.g., CPU utilization, Disk Read Bytes, etc., is done by meters or counters. Typically there is a meter for each resource being tracked, and there is a separate meter for each instance of the resource. It is important to note that the lifetime of a meter is decoupled from the associated resource, and a meter continues to exist even after the resource it was tracking has been terminated [46]. Each data item collected by a meter is referred to as a "sample," and consists of a timestamp to mark the time of collected data, and a volume that records the value. The polling interval between two events is specified in the pipeline.yaml file and can be adjusted according to the cloud provider requirements. Once configured, the same polling interval is used to monitor all the instances launched in that particular setup, unless the cloud provider manually changes it.

3.3 Open Problems in Cloud Monitoring

In this section we present our analysis of how monitoring and metering has been implemented in OpenStack and pinpoint existing problems.

3.3.1 Constant polling frequency

Category	# of VMs	Duration
AP	737	3 months
Africa	400	8 days
Australia	999	3 months
EMEA	1223	3 months

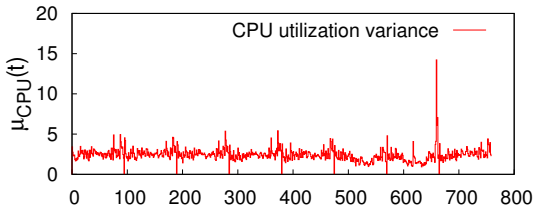
Table 3.1: Range of IBM production servers used in our study.

The frequency at which samples are collected for a certain meter is called the polling frequency for that meter. OpenStack allows cloud service providers to manually configure the polling frequency for different types of meters, however, once configured it remains constant unless updated manually. Hence, cloud service providers end up collecting large amounts of close to identical samples often carrying information of low significance about the usage or state of the tracked resource. This results in high storage volume and increased computational resource requirements to first collect and then process the collected data in view of metering, incident, or problem management purposes, to name a few. To understand the problem and potential solutions, lets assume a concrete case where the utilization of a certain resource remains constant at a specific value for 5 hours. Assuming the polling frequency is per second, we collect in total 18000 samples. From the metering point of view this same information would have been inferred from data collected according to a polling frequency of one or a few samples [1] per hour. If we scale this calculation to hundreds of metrics on thousands VMs in a typical cloud setup, the problem increases by many folds.

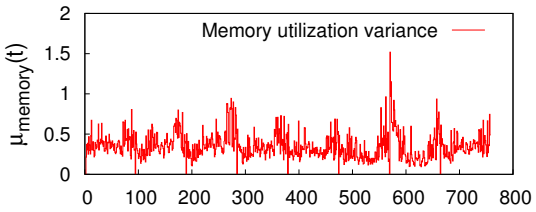
The logical question to ask is how prevalent is in a typical cloud environment this scenario where the resource utilization remains unchanged? To answer this question we collected and analyzed the data from 3359 machines launched in geographically distributed IBM production servers. Table 3.1 shows the range of IBM production servers used in our study, in Asia Pacific, Africa, Australia, and Europe. The data was collected over a period of 3 months with a sample collected after every 15 minutes.

$$\left|\frac{dRvm_i(t)}{dt}\right| = |Rvm_i(t) - Rvm_i(t-1)| \tag{3.1}$$

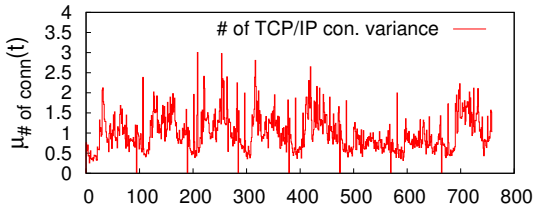
[1] Cloud providers may be interested in collecting more samples to properly track the health of the monitored resource. We will address this case later.



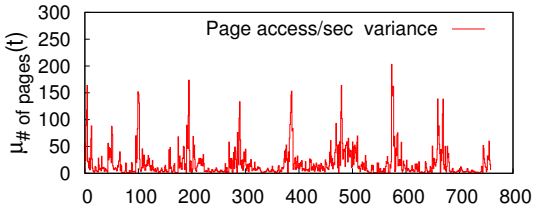
(a) CPU utilization



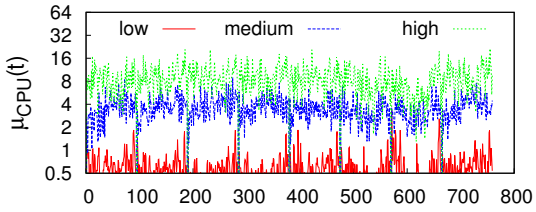
(b) Memory utilization



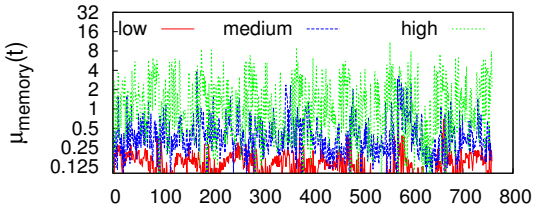
(c) # of TCP/IP connections



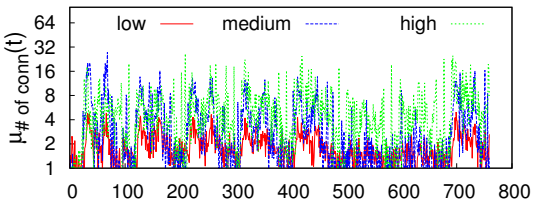
(d) Pages accessed per second



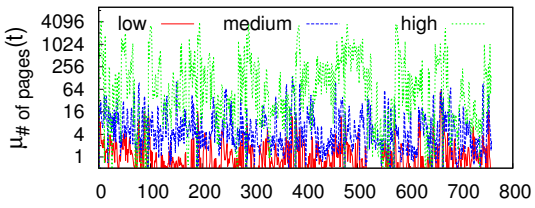
(e) Tiered 'CPU utilization'



(f) Tiered 'Memory utilization'



(g) Tiered '# of TCP/IP con.'



(h) Tiered 'pages accessed per sec.'

Figure 3.1: Mean of absolute rate of change in utilization of resource R, i.e. $\mu_R(t)$, for studied VMs; and tiering of VMs in three different clusters based on rate of change in utilization for resource R, i.e. $\left|\frac{dRvm_i(t)}{dt}\right|$.

$$\mu_R(t) = \frac{\sum_{i=1}^{N} \left|\frac{dRvm_i(t)}{dt}\right|}{N} \tag{3.2}$$

To study the variance of resource utilization, we calculated the mean of absolute rate of change, $\mu(t)$, for different monitored resources of randomly picked 338 VMs from all the regions. We analyzed two different kinds of meters: i) Meters used to directly monitor the infrastructure usage like CPU and memory utilization, and ii) Meters used to track the load imposed by VMs on the physical infrastructure like number of TCP/IP connections established by VM and pages accessed per second from the disk. We first calculated the absolute value of the rate of change at time t in each resource usage (e.g., CPU utilization, memory utilization, number of TCP/IP connections, pages accessed from disk/sec. etc.), for each VM i, as shown in equation 3.1. Then we average across all VMs as shown in equation 3.2.

Figure 3.1(a) and 3.1(b) show $\mu(t)$ of the CPU and memory respectively for the last 8 days of our collection of data. We found over this period of 8 days the instantaneous variation of the resource usage to be less than 5%. We repeated this evaluation on the two other type of meters, '# of TCP/IP connection' and 'pages accessed/sec'. We found the same trend for meter monitoring as shown in figure 3.1(c) and 3.1(d). This behavior hold true in average across VMs and the 3 month of data in our collection. Overall we found less than 5% of VMs having sudden variation in the tracked resource utilization.

An additional observation is that for the 5% of VMs having sudden variation in resource utilization, the polling was not frequent enough to properly capture the evolution of the change. Hence, if on one hand decreasing the polling frequency can be beneficial for the majority of the VMs, on the other hand it is advantageous to identify those VM that benefit from an increased polling frequency compared to the default in order to better capture their behavior in view of modeling it.

3.3.2 Global polling frequency for all VMS

For each meter, Ceilometer uses the same polling frequency globally across all the VMs launched in an OpenStack setup. Tiering based on variation in resource utilization can enable a cloud service provider to monitor and collect samples from each tier at a different polling frequency, hence allowing to track at a lower polling frequency the resources having less resource usage variation. Let us consider the scenario where a metric exhibits 3 types of behaviors across a set of VMs as follows: high variance in utilization of monitored resource; predictable or medium variance; and low variance.

Currently, the cloud providers are limited to monitor this resource by sampling it at one and the same frequency in all VMs, which would be the polling frequency to capture a predefined volume of changes in the utilization of that resource, e.g., 98%,. This limitation of choice in the sampling frequencies leads to resources in the low variance tier to be over sampled by being monitored at a too high frequency, whereas the resources in the high variance tier are undersampled.

To further understand the characteristics of our data, we manually divided the VMs into three different tiers by calculating mean rate of change in resource utilization for each of the 338 VMs. We defined thresholds by dividing the range between maximum and minimum found values in three equal tiers for each of the four resources. Following the levels of variance in $\mu_R(t)$ of the four monitored resource, we found that for each meter used to track these resources, more than 80% of the VMs fall in the tier with low variance; 15% were found in the tier with medium variance and 5% in the ties with high variance. Figure 3.1(e) and figure 3.1(f) show the results for tiered CPU and tiered memory utilization, respectively. Similarly, figure 3.1(g) and figure 3.1(h) show the results for tiered '# of TCP/IP connections' and 'pages accessed/sec', respectively. This analysis shows that in case of un-tiered sampling, 80% of resources were monitored at a frequency higher than needed to capture their changes, whereas 5% of the resources were monitored at a frequency smaller than needed to capture their changes.

3.3.3 Lack of policy based monitoring

To enable custom, tier based polling frequencies, OpenStack should support a policy based data monitoring. A monitoring policy defines the sampling rules for a given metric profile. Currently, Ceilometer lacks this feature, thus prohibiting the cloud providers from defining metric profiles and associated differentiating policies.

The applications hosted by a system dictate their monitoring and data retention requirements. For instance, the monitoring requirements for desktop clouds are different from those for HPC application or for MapReduce jobs. Similarly, monitoring and data retention requirements for charging purposes are different from those for system health check. Metrics with higher (/lower) monitoring data resolution requirements should be able to configure in their profile that their data is critical (/not critical) and hence an conservative (/reduced) sampling and storing policy can be associated to them. Another advantage of policy based sampling is that it enables cloud providers to separate those resources for which prompt anomaly detection is required.

Furthermore, what monitoring data needs to be collected and how monitoring data is collected highly depends on usage of these monitoring data. Our goal is to enable the customization of metric profiles by allowing the providers to characterize each metric in terms of what it is used for (e.g., used for charging the tenant as part of the usage based price definition, used for health check etc.), or in terms of its importance in the inventory (e.g., belongs to VM hosting critical or non-critical applications), in terms of its precedence in the application flows (e.g., belongs to a leaf or root item in the dependency graph), or in terms of its dependence on other metrics (e.g., independent or correlated metric). These criterias, while extendable, represent key configuration items to be defined in the metric profile. Based on the metric profile, each metric is associated a metric policy. A polling policy can be as naive as collecting and storing only the data from the last hour, day or even week initially

Metric Profile				Metric Policy
Critical	Dependency root	Used for usage based charging	Independent metric	
✓	∗	∗	∗	Conservative sampling and storage
✗	✓	∗	∗	Conservative sampling and storage
✗	✗	✓	∗	Conservative sampling and aggregated storage [a]
✗	✗	✗	✓	Per tier sampling and conservative storage
✗	✗	✗	✗	Per tier sampling and aggregated storage

Table 3.2: Example of mapping between metric profile and metric policy (∗ means either of ✓or ✗).

[a]Replaced by conservative storage in case the provider is required to keep evidence of the raw metered data for financial regulations.

at full granularity and then aggregate over time; or it can be as complex as collecting data, storing it, analyzing it and then fine tune how to capture and store it in a more efficient way.

3.4 System Design

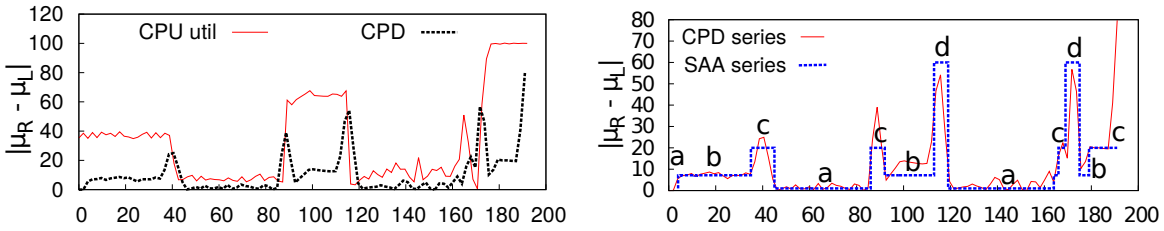
In this section we present a methodology which enables the cloud providers to customize their service management monitoring system for a policy based data monitoring such that each tier of systems with similar monitoring data behavior and business needs gets its VMs monitored according to the same dedicated polling policy, different from the other tiers.

3.4.1 Metric profiling

Our solution allows cloud providers to specify configuration items such as: (i) REST URLs of the usage calculation classes for the usage based pricing -these classes typically calculate the transformation of the raw metered data into the charged unit of measure, or the maximum or sum of the metered values, or more complex metric aggregations; (ii) inventory and interdependencies between servers, applications, network devices, software, configuration files, operating systems and other IT infrastructure components expressed as graphs, xml files or spreadsheets [30]; (iii) event correlation engine API [29].

The proposed solution programmatically and periodically accesses the specified sources of data to automatically populate or update the profile of each metric. A mapping between each item in the profile and its corresponding sampling rule is to be maintained manually by the cloud provider. For instance, a metric that is critical or belongs to root item in the dependency graph requires a more conservative sampling and storage policy compared to a metric that is non-critical, belongs to a leaf item in the dependency graph or is not used for usage based charging.

Table 3.2 illustrates a potential mapping between metric profiles and metric policies. A



(a) CPD (b) SAA

Figure 3.2: Change Point Detection and Symbolic Aggregation Approximation on CPU utilization of a VM over period of two days.

conservative sampling implies using the default high frequency sampling of the monitoring system; a conservative storage means recording all the collected data samples; a per tier sampling denotes a reduced sampling frequency as inferred by our solution for the behavior of the metrics in that tier; an aggregate storage means applying a corresponding aggregation rule e.g., keeping the independent metric only in case of correlated metrics, or aggregating as indicated by the usage calculation class for a metered metric, etc., and then recording only the value of the aggregate.

3.4.2 Tiered polling frequency

Once configured and enabled, the service management monitoring system, e.g., Ceilometer, polls the systems metrics initially at a regular, default frequency. We first identify the metrics suitable for aggregated storage and apply the aggregation technique on the fly. For instance, for metrics characterized as correlated to another metric, we compare the new sample to the last stored sample and store the new one in the database only if its value is different than the last one. The metrics used for usage based charging will be applied the usage calculation logic (e.g., sum, max, min) and the result will update the current value without creating a new entry in the database.

Then we identify the metrics suitable for tiered sampling. Each sample of data collected from the cloud environment is a data point in the time series of that metric on a given configuration item. After sufficient data is collected, e.g., for a few weeks, we apply a Change Point Detection (CPD) analysis algorithm [63] on their time series to identify the number and timing of the changes that occur in each metric. Figure 3.2(a) shows an example of CPD applied on a CPU utilization of a VM over period of two days. We subsequently apply Symbolic Aggregate Approximation [54, 141] on the change point time series to convert the data into a discrete format of a sequence of symbols with a small alphabet size as shown in figure 3.2(b). The strings obtained, e.g. 'abcacbdacdbc' in given example, encode the change behavior of the metrics selected for a reduced sampling. Thus, one symbol corresponds to

the flat, monotone segments of invariable behavior, while a few other symbols indicate when the metric changed. Therefore, we are interested to collect precise data around the timing of the change occurrences which correspond to changes in the metrics statistical properties, while collecting samples during the monotone stretches of consecutive identical symbols is of less interest. When these segments of unchanged performance keep for many hours, it is unnecessary to poll and store data at 1 to 5 seconds interval. In consequence, a few samples only are to be collected during those segments of unchanged performance.

Furthermore, we notice that large groups of metrics have similar change point time series, with spikes corresponding to seasonal periods of the day and of the week/week-end. To reduce the number of metric policies to maintain, we split the sequences in segments corresponding to weekly periods (Monday to Sunday) and group them. We employ the Structural Similarity algorithm [54, 141, 156] to cluster the symbol segments generated above into tiers of metrics with similar sequences. A metric having all its weekly segments in one group indicates that its weekly pattern is stable across the analyzed weeks. We filter out of the tiers all the metrics with segments scattered in different clusters. We infer then the polling frequency for the remaining metrics in a tier from their weekly sequence of symbols as follows:

- For each isolated occurrence of a spike symbol in any metric segment, data is collected from the timing of the beginning of the spike until the end of the spike (e.g., every second during a minute).
- For the segments with unchanged performance, data is collected hourly only if no isolated spike has already triggered data collection during that hour.

The signature of the polling timing identified in each tier represents the sampling metric policy of that tier. As these policies are made available by our solution, the service management monitoring system can switch to from the default policy and start making an efficient usage of storage and computational resources.

Online Analysis: as a metric behavior or monitoring policy evolves over time, its classification in a particular tier may become unsuitable for the new sampling and storage requirements. A metric policy update will directly trigger the reclassification of the metric into the default, full sampling tier, where the process of classification described above will be re-applied. However, a metric behavior update is not directly signaled unless it is monitored. To this end, we compare the samples collected for tier based sampling to the most recent average of the past collected values corresponding to the metric's sequence symbol generated by the Symbolic Aggregate Approximation for that particular time in the sequence. If the difference is bigger than a threshold (e.g., 10%), we place the metric into the default, conservative sampling tier, for reclassification. An additional goal is to identify and use in the meter profile those configuration items that have a reduced sensitivity to the changes in the monitored environment, and hence a limited potential of causing the metrics to oscillate between the tiers.

3.5 Preliminary Results

We used Python and R [51] for this preliminary evaluation of our solution. The data analyzed was collected from IBM production servers over a period of 3 months (Table 3.1 and §3.3.1).

Policy	Data Reduction		Miss Detecion Rate
	Non Aggressive	Aggressive	
Cons_Sam + Cons_Stor	0 %	0 %	3 %
Cons_Sam + Aggr_Stor	61.17 %	70.17 %	3 %
Tier_Sam + Cons_Stor	72.7 %	76.8 %	0.05 - 0.1 %
Tier_Sam + Aggr_Stor	76.32 %	80.04 %	0.05 - 0.1 %

Table 3.3: Data reduction and miss detection rate under different policies.

For evaluation purposes we compared default or conservative sampling and storage with tiered sampling and aggregated storage. The policies introduced in §3.4.1 and Table 3.2, were defined based on two procedures, one aggressive and the other non aggressive. In the aggressive approach, i) we set a higher threshold for aggregated storage, and ii) we aggressively reduce the polling frequency when collecting monitoring data from tiers with low variance in the resource utilization. For each policy, we measured the reduction in the collected data size, as well as the missed anomaly detection rate. The missed anomaly detection rate was calculated by comparing the data collected for each policy with the anomalies found by examining the system logs collected for the same time period using sysstat utilities [58]. The sysstat data was collected at a frequency double than the maximum polling frequency used to collect the monitoring samples. We defined anomaly as a missed sample having a sudden increase or decrease in utilization as compared to its adjacent samples. Furthermore, for the policies involving metered data we enforced that we store enough samples so that metering tasks can be successfully performed in following revenue calculation stages.

Table 4.1 illustrates the results of our evaluation. Notice that by storing aggregated data instead of conservatively storing all samples, we obtained a 60% to 70% reduction in data collected for monitoring purpose. The decrease in data size was due to storing only samples which either conveyed useful information about the current health of the monitored resource or were required for charging purposes. The missed anomaly detection rate for conservative sampling was found 3% as the default polling frequency was not high enough to track the changes in VMs having sudden variations in resource utilization.

Next, when tiered sampling was enabled with conservative storing of every sample, we were able to reduce the data size by 72% to 76% whereas missed anomaly detection rate was found to be only 0.05% to 0.1% due to higher polling frequency used for the set of VMs having sudden variations in resource utilization. Further evaluation revealed that 99.99% of the anomalies were from the tier for which reduced the polling frequency was used. Hence missed anomaly detection rate can be further reduced by using less aggressive approach.

Finally, when applying both tiered sampling and aggregated storage, we obtained up to 80%

reduction in data size. Missed anomaly detection rate remained the same as in the case of tiered sampling with conservative storing, since shifting from conservative to aggregated storage without changing the sampling policy does not affect the anomaly detection.

3.6 Potential Impact and Current Work

We estimated the storage savings by considering an average object size of a sample of 1024 bytes [53]. This size is due to the information related the resource usage plus the additional fields, e.g., instance_id, timestamp, resource_id, user_id, project_id, etc. If a single VM produces 100 counters per second and storage costs $0.07 per GB then a rough estimate of savings for an environment of 1000 VM, per year can be calculated as following:

0.8 x $0.07 per GB / month x 100 samples / sec x 1024 bytes x 60 sec / min x 60 min / hour x 24 hours / day x 30 days / month / 10 ^ 9 bytes / GB = $14.864 / VM / month x 1000 VMs = $14,864 / environment / month, which accumulated over one year results in a saving of $1,159,392 / environment / year [2].

It is important to note that in some cases (e.g. metering data for charging) cloud providers are bound by SLA to keep customer data for as long as 3 to 5 years. Therefore, our solution is beneficial to both tenant (for the savings) and provider (for the competitive advantage).

On-going efforts We are looking for new criterias of evaluation besides the missed anomaly detection that could shed additional light on the impact of our policy based monitoring on the service management functionality. Also, the current solution attributes fixed weights to the profile items in the different policies, while a change in the environment, e.g., an incident, may increase the relevance of leaf systems detailed monitoring data and hence require an augmented weight to be reflected in an increased polling frequency. Furthermore, we are evaluating our solution in real time scenarios. Our plan is to have our solution integrated with IBM production servers to directly evaluate it by comparing it with existing more conservative setups used to collect monitoring data. This will help in gaining more confidence on our approach and carve the aspects that lead to positive results. We are also investigating the effects of using different window sizes for CPD analysis, and performing more fine grained symbolic approximation of data generated by CPD, on the resource requirements for cloud management. This evaluation will provide to the cloud providers further control on the monitoring setup as it will enable them to perform informed tradeoffs between the degree of monitoring and the cost associated with the monitoring setup.

[2] The amount of stored data increases each month and can be represented by an arithmetic progression. Hence, $n\frac{(a_1+a_n)}{2}$ was used to calculate the cost accumulated over period of 12 months.

3.7 Summary

In this work, we presented an analysis of how monitoring data is collected by OpenStack. By analyzing the characteristics of the collected monitoring data, we identified several open problems and presented alternative solutions. Our preliminary evaluation using actual data from IBM production servers reveals that it is possible to reduce the monitoring data size upto 80% and missed anomaly detection rate from 3% to as low as 0.05% to 0.1%.. We believe that we have looked deeply into how monitoring subsystem provided by open source cloud solutions collect data and this work will lead to lots of discussion on how these subsystems can be improved.

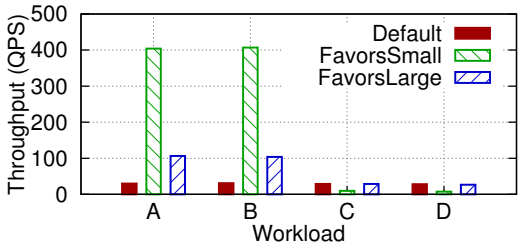
Chapter 4

Resource Partitioning of Cloud Storage

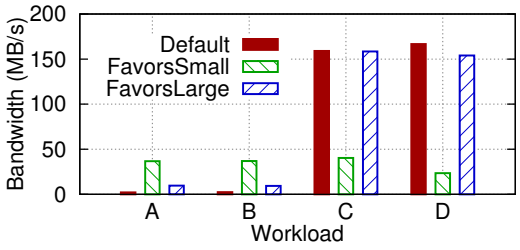
4.1 Introduction

Cloud object stores, such as Amazon S3, Google Cloud Store (GCS), OpenStack Swift and Ceph [166], have become the most widely used form of cloud storage in recent years. These stores combine key advantages such as high availability, elasticity and a "pay-as-you-go" pricing model, which allows applications to scale as the usage increases or decreases, and offers HTTP-based RESTful APIs for easy data management. The desirable features, coupled with the advances in virtualization infrastructure, are driving the adoption of cloud object stores by a myriad of applications. Examples range from web applications [80] that store image and video files, to backup services [28] that require large capacity for archival data, to big data analytics frameworks [95]. Similarly, object stores are increasingly being adopted by the HPC community [115] as they provide efficient metadata management and scalability that helps in extreme-scale high-end computing, and allows for seamless adaptation to a wide range of general purpose and scientific computing file system workloads [166].

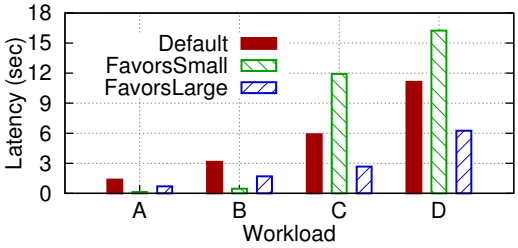
A typical deployment of cloud object stores either opts to use a monolithic configuration or segmented storage setup [57] with a static configuration to handle different types of applications with evolving requirements. Using a monolithic configuration setup results in all applications experiencing the same service level, e.g., similar average latency per request, data transfer throughput, and queries per second (QPS). However, different applications entail extremely different latency and throughput requirements. For example, a social networking or photo sharing application requires low latency to support a highly-responsive user experience, whereas backup services can tolerate higher latency but require sustained high throughput. Some object stores provide static configuration of storage policies, e.g., Swift segmentation [57], to allow for segmenting the cluster. This static storage segmenta-



(a) Throughput (QPS)



(b) Bandwidth (MB/s)



(c) Latency (s)

Figure 4.1: Performance achieved under various object store configurations in a multi-tenant environment.

tion policy is limited to cover only the storage server layer of an object store. However, a typical object store setup also consists of additional layers such as proxy server and load balancer layer. End-to-end performance of an application depends on correct configuration at all these layers to meet its requirements, and not only the storage server. Thus, a comprehensive solution is needed.

From the cloud provider's perspective, supporting dramatically different workloads from different applications (tenants) using a single homogeneous configuration means that optimization opportunities are lost. Each different application represents a workload with different characteristics. For example, a photo sharing application such as Instagramwould have a large number of small-medium sized files (e.g, KB- to MB-level image objects), with skewed access pattern where frequent read and write requests go for hotter/popular objects. In contrast, an enterprise backup application (e.g., Arq [3]) consists largely of write requests for large cold archive files with reads only sparsely arising. Using a homogeneous configuration prevents fine-tuning of the system to such varied needs and reduces overall system efficiency.

The situation is further complicated by the fact that due to regular system upgrades and introduction of new storage architectures, data centers hosting the object stores are becoming increasingly heterogeneous [107, 145]. However, with either the "one-size-fits-all" monolithic deployment or static storage segmentation policy driven partitioning, it is impossible to match specific types of hardware with the right type of application workload. For example,

latency-sensitive small-object workloads would require low-latency storage devices and powerful CPU processing capacity, whereas large object write-only workloads can be supported with a combination of high network bandwidth across all layers (e.g., load balancer, proxy, and object servers etc.) and weaker CPU power. Under these scenarios, meeting SLA requirement for one of the workloads may require, (i) adding hardware resources that may not improve the performance for other workloads, and (ii) software tuning that may decrease the performance for other workloads.

Furthermore, the workloads seen by the object store are varied and fluctuate over time. Consider a scenario where the workload demand from one application (tenant) is spiking while the demand from another application that shares the same object store resources is dipping. In this situation, static policies need to be updated based on the changes experienced in the workload. This calls for a new object store architecture that can dynamically perform resource provisioning for driving online reconfiguration across multiple partitions of the object store.

Hence in this work, we posit that compared to using a rigid object store *it is more beneficial to support multi-tenant workloads separately using dynamically configurable finer-grained object stores* on sub-clusters of available resources.

Motivational Study. To motivate our approach and demonstrate the need for differentiated object stores, we study different types of representative practical workloads as follows. We examine four different real-world applications that use cloud object storage as listed in Table 4.1. We deploy and evaluate OpenStack Swift in a multi-tenant environment using COSBench [172] as workload generator configured for the four types of studied workloads. Swift is a popular object store implementation provided by OpenStack that is increasingly becoming the *de facto* cloud computing software platform. In these tests, we use three different Swift configurations (setups)[1]. We run COSBench clients on designated machines to saturate Swift. Each benchmark is run for 15 minutes after all data is loaded into the store. We use two nodes as proxy servers in each of the configuration. To simulate datacenter heterogeneity, one of the proxy server has 32 cores while the other has 8 cores. The proxy server running on the 32-core machine is connected to the storage nodes via 1 Gbps interconnect, while the proxy server on the 8-core machine is connected via 10 Gbps interconnect. In addition, four 32-core machines are used as storage nodes. Each storage node has 3 SATA SSDs. The storage nodes are well-endowed and configured in such a way so as not to become a performance bottleneck for any of the studied configurations.

[1]The integrated Swift storage policies only support static storage node segmentation. In our motivational study we keep the storage node settings fixed and vary the proxy node settings. We do this to highlight the need for a more comprehensive workload-aware scheme, which is the subject of this work.

Workload	Workload characteristics		App. scenario
	Obj. size	Operation distribution	
A	1–128KB	G: 90%, P: 5%, D:5%	Web hosting
B	1–128KB	G: 5%, P: 90%, D:5%	Online game hosting
C	1–128MB	G: 90%, P: 5%, D:5%	Online video sharing
D	1–128MB	G: 5%, P: 90%, D:5%	Enterprise backup

Table 4.1: Different types of workloads and application scenarios used for testing the behavior of object stores. G: GET operation; P: PUT operation; D: DELETE operation.

Default configuration: The default monolithic Swift setup is used where both 8-core and 32-core machines acted as proxy server. The workloads are handled by all resources and round robin DNS was used to distribute the requests to the proxies.

FavorsSmall configuration: The available resources are divided into two sub-object stores, one configured for workloads with small objects and the other for large objects. One 8-core machine (connected via 10 Gbps) served as proxy for `WorkloadA` and `WorkloadB`, and one 32-core machine connected via 1 Gbps network served `WorkloadC` and `WorkloadD`.

FavorsLarge configuration: One 32-core machine (connected via 1 Gbps) is used as proxy for `WorkloadA` and `WorkloadB`, while one 8-core machine (connected via 10 Gbps) is used as proxy for `WorkloadC` and `WorkloadD`.

Figure 4.1 shows the comparison of performance achieved under the studied configurations. As shown in Figure 4.1(a), separating proxy servers for different workloads improved the overall QPS by 700% and 225% for `FavorsSmall` and `FavorsLarge`, respectively, compared to the default Swift setup. It is interesting to note that even though `FavorsSmall` resulted in very high QPS for small objects of (`WorkloadA` and `WorkloadB`), it is not the best configuration as it significantly affects the MB/s (dropped by from 350% to 500%, as observed in Figure 4.1(b)) for workloads dominated with large object (`WorkloadC` and `WorkloadD`). On the other hand, in `FavorsLarge` the throughput for large objects remained same. Similarly, the latency of `FavorsLarge` is also less than that achieved by the default configuration for all the workloads (Figure 4.1(c)). `FavorsSmall` provides best and worst latency for small and large object workloads, respectively. We also observe that switching to different network connections on proxy servers in `Default` configuration results in similar results. These results demonstrate the need for a comprehensive study of the impact of different configurations on performance to ensure efficient cloud object store design.

From our experiments, we infer the following. (i) Cloud object store workloads can be classified based on the size of the objects in their workloads. In case of small objects, cloud tenants are mostly interested in QPS and latency, whereas for large objects data throughput is considered more important. (ii) When multiple tenants run workloads with drastically different behaviors, they compete for the object store resources with each other, the workload

dominated with small objects experiences a dramatic loss in performance. This is because the available network bandwidth is exhausted to transfer TCP packets containing payload for large objects, hence wasting the CPU power that would have been utilized to serve workloads with small objects on object storage nodes. That is why using a separate proxy server under `FavorsSmall` and `FavorsLarge` gives a fair chance to small object workloads to be properly handled by the storage nodes. Thus, cloud object stores need better resource management to ensure that tenants are treated equally.

Contributions. To this end, we propose MOS, a novel micro object storage architecture with independently configured micro-object-stores each tuned dynamically for a particular type of workload. We then expose these microstores to the tenants who can then choose to place their data in the appropriate microstore based on the latency and throughput requirements of their workloads. We further enhance our basic resource provisioning engine to build MOS$_{++}$, which incorporates the container abstraction for fine-grained resource management, SLA awareness, and better resource efficiency.

Specifically, we make the following contributions:

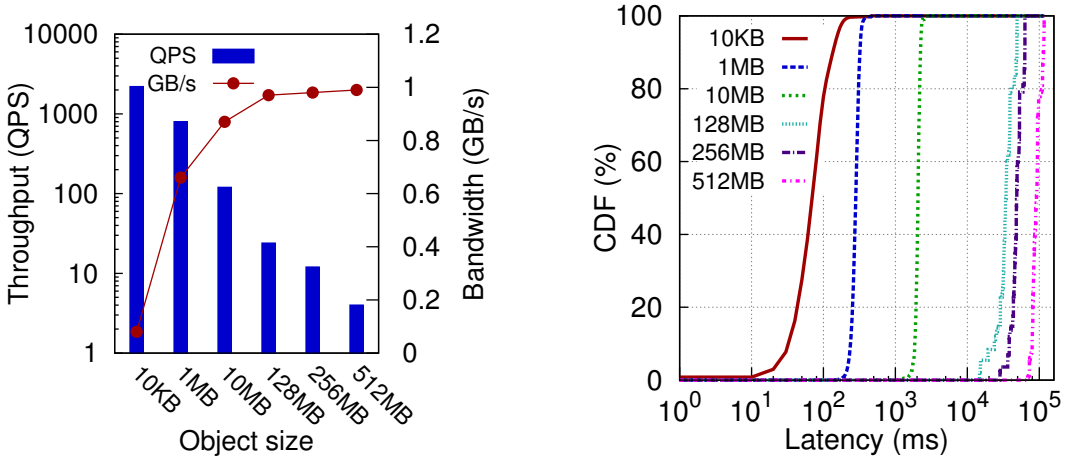
- We evaluate the impact of conventional object storage configuration on performance and resource efficiency by conducting experiments on a local Swift testbed. Our observations stress the need to carefully evaluate the various configuration choices and develop simple *Rules-of-Thumb* that cloud providers can leverage for provisioning the object storage.

- We perform a detailed performance and resource efficiency analysis on identifying major hardware and software configuration opportunities that can be used to fine-tune object stores for specific workloads. Our findings indicate the need to re-architect cloud object storage specialized for the public cloud.

- Based on our behavior analysis, we design MOS, an object store that (i) dynamically provisions microstores, each configured with different combination of hardware and software options, and (ii) exposes the interfaces of microstores to the tenants to use according to application requirements.

- We extend our basic framework to MOS$_{++}$ that uses container based approach to launch resources in a more fine-grained manner. MOS$_{++}$ is SLA-aware, supports rapid deployment, portability across machines, offers a lightweight footprint, and simplifies maintenance.

- We implement a prototype of MOS$_{++}$, and demonstrate that our approach results in improved performance (by up to 89.6% and 79.8% compared to the default monolithic and statically configured object store setup, respectively), as well as higher resource



(a) Read performance.

(b) Read latency distribution.

Figure 4.2: Impact of varying the object size on read performance. Note the log scale on the Y-axis of Figure 4.2(a) and the X-axis of Figure 4.2(b).

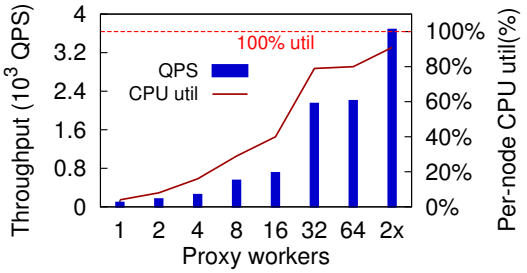
efficiency. Furthermore, we design a simulator to evaluate our solution under a large-scale 456-core cloud cluster setup. We also compare the performance of MOS with MOS$_{++}$ to highlight advantages of our container based approach.

4.2 Analysis

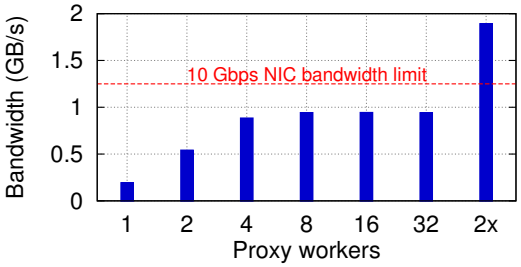
In this section, we present a detailed analysis of how object store behaves under various software and hardware configurations. Next, we use the study to develop *rules-of-thumb* for configuring object stores, which guide the design of MOS.

In the following analysis, we use a 32-core machine as a proxy node with two 32-core storage nodes each equipped with 3 SSDs (to eliminate the storage bottleneck), unless mentioned otherwise. For workloads dominated by small objects (at KB level) the metrics of interest are *throughput* in terms of queries per second (QPS) and response latency, while for workloads dominated by large objects (at MB–GB level), *bandwidth* in terms of MB/s or GB/s is more important.

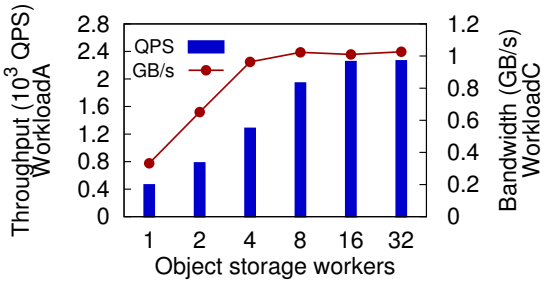
Q1: How does object size impact performance? First, we analyze the impact of object size on performance in terms of throughput (QPS) and bandwidth (GB/s). While QPS captures the object-wise throughput performance, the bandwidth serves as an important metric reflecting byte-wise performance. As shown in Figure 4.2(a), increasing the object size



(a) Effect of varying proxy capability (A).



(b) Effect of varying proxy capability (C).



(c) Effect of varying object store server capability.

Figure 4.3: Studied software/hardware configuration options. In Figure 4.3(c), small-object workloads refer to bars (QPS) while large-object workloads refer to linepoints (GB/s). A: `WorkloadA`; C: `WorkloadC`.

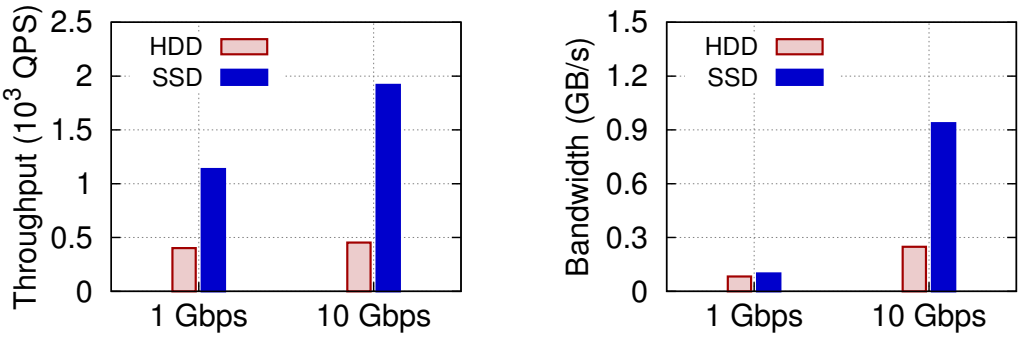
results in the throughput decreasing drastically. Specifically, when the object size is increased from 10 KB to 10 MB, we observe the increasing tendency of the network bandwidth. When the object size is increased further to above 128 MB, the bandwidth only improves marginally (from 0.97 GB/s to 0.98 GB/s), implying that the NIC is saturated. Figure 4.2(b) plots the corresponding latency distribution at each studied object size. At large object sizes (10 MB–512 MB), the request response latency is more than 100× than that for small object sizes (10 KB–1 MB). From these tests, we can infer that, as long as the object size exceeds a certain threshold, network bandwidth becomes the limiting factor. Correspondingly, this again, explains why `WorkloadA` and `WorkloadB` achieve extremely poor performance when co-existing with `WorkloadC` and `WorkloadD` in §4.1. Hence, the tests demonstrate that, in a multi-tenant environment with mixed workloads, individual workloads should be partitioned and serviced through disjoint object stores to reduce mutual interference and performance impact.

Q2: How does proxy server configuration impact performance? Next, we study the effect of scaling proxy nodes on workload performance. We vary the computational capacity of the proxy node by increasing proxy's allotted CPU cores. Figure 4.3(a) shows the proxy

tuning effect. As we increase the proxy workers in one proxy node the QPS is improved linearly until we reach 32 proxy workers. The observed CPU utilization reaches close to 85% (bounding the throughput) with both 32 and 64 proxy workers, implying that CPU becomes the bottleneck here. Adding one more proxy node (`2x`) almost doubles the performance (QPS increased from $2,200$ to $3,700$), clearly demonstrating that proxy's performance is constrained by the CPU capacity. Next, we repeat the test with large object workloads. As shown in Figure 4.3(b), the network bandwidth limit is reached as soon as the number of proxy workers reaches 4, with modest CPU utilization (about 25%) observed on the proxy node. This is because for large object workload, the performance becomes constrained by the network bandwidth before CPU can be saturated. Hence adding another proxy node (`2x`, i.e., doubling the available network bandwidth) results in linear increase in throughput. Thus, the takeaway is that a proxy's computational capacity can act as the bottleneck for workloads dominated with small objects, whereas the network bandwidth is the limiting resource for workloads dominated by large objects.

Q3: How does storage server configuration impact performance? Next, we study the effect of scaling object storage nodes on workload performance. As shown in Figure 4.3(c), the peak QPS for small object workloads is achieved with 16 object storage workers, which is exactly the same as the number of proxy workers launched to achieve this QPS (recall that two object storage nodes are deployed behind one proxy server node). This implies that the maximum performance can only be achieved when both the proxy and storage nodes are equipped with the same amount of CPU resources, which strengthens our observation that CPU capability is the limiting factor for small-object workloads. In contrast, for large-object workloads, the network limit is quickly reached with only 4 object storage workers. This is because, for large objects the performance is bottlenecked by the network (recall that each storage node has 3 SATA SSDs, thus disk bandwidth does not pose a limitation in our test).

Q4: How does network/storage affect performance? In our next test, we study the effect of varying storage device and network connectivity on workload throughput. Figure 4.4(a) shows that faster network interconnect (1 Gbps NIC $\rightarrow$ 10 Gbps NIC) results in only 12% increase in QPS for small object workloads with HDD as storage medium, and 70% increase when SATA SSD is used. This observation shows that small-object intensive workloads are more sensitive to the storage devices rather than the network bandwidth. Thus, they may be efficiently handled using a lower-bandwidth network interconnect but by using high-bandwidth storage devices. On the other hand, increasing network interconnect improves performance by as much as 900% (using SSDs) in case for large-object intensive workloads (Figure 4.4(b)), which clearly indicates such kind of workloads can benefit from high-bandwidth network interconnects.



(a) `WorkloadA`. (b) `WorkloadC`.

Figure 4.4: Performance of the object store equipped with **homogeneous storage devices** as a function of the NIC bandwidth.

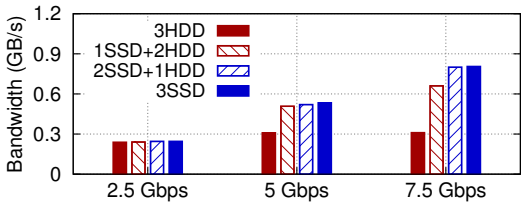


Figure 4.5: Performance of large-object workload with **heterogeneous storage devices** as a function of the NIC bandwidth. The corresponding bandwidth (GB/s) with 2.5 Gbps, 5 Gbps, and 7.5 Gbps are 0.31 GB/s, 0.62 GB/s, and 0.93 GB/s, respectively.

Q5: What is the impact of heterogeneous storage setup on performance of large-object workloads? Finally, we study the impact of heterogeneous storage configuration on large-object intensive workloads. Here, we limit the network bandwidth using Linux traffic control tool `tc`. We measure the performance of the object store under a large-object workload, with four setups: two heterogeneous setups 1 SSD + 2 HDD and 2 SSD + 1 HDD; and two homogeneous setups 3 HDD and 3 SSD as baselines. Figure 4.5 demonstrates that the choice of different storage device type combination changes based on the network bandwidth limit. We vary the network bandwidth limit to emulate the scenario where the network is partitioned in a multi-tenant environment. Note, when the network is limited to 2.5 Gbps, all four storage configurations achieve the same performance. Thus, the storage setup of choice under 2.5 Gbps is 3 HDDs. As the bandwidth limits increases to 7.5 Gbps, the 3 HDDs setup becomes the worst choice, especially when meeting SLAs is critical. Here, the 2 SSD + 1 HDD setup is desirable as it achieves almost the same performance as the 3 SSD setup, but with a higher resource efficiency. These tests necessitate the need for a workload-aware resource provisioning mechanism that selects the most efficient and high

performing options under dynamically changing workloads and tenant requirements.

Summary of Rules-of-thumb. It is fairly straightforward to manually tune the object stores by controlling all the other configuration variables. However, it is a challenging task to dynamically detect the workload shifts and meet the tenant goals while maximizing the resource efficiency at runtime, particularly when the service providers are faced with many software and hardware configuration options. To this end, we develop the following *rules-of-thumb* that are helpful in guiding the online/offline performance tuning of object stores as well as the design of MOS.

R1 Cloud object store design can benefit from (i) partitioning the monolithic object store architecture based on workload characteristics, and (ii) separately servicing interfering workloads in the multi-tenant environment. Object size distribution is a key factor for classifying workload characteristics.

R2 CPU capacity of proxy servers is the first-priority resource for small-object intensive workloads. CPU becomes a bottleneck much earlier than the network for such workloads.

R3 On the other hand, availabile network bandwidth plays a critical role in the performance of large-object intensive workloads.

R4 The number of CPU cores used in storage nodes can be safely configured based on the number of deployed proxy workers, given that the storage devices provide sufficient disk bandwidth. This rule can be modeled using the following equation: $proxyCores = storageNodes * coresPerStorageNode$. E.g., one 32-core proxy node may require four 8-core storage nodes.

R5 The aggregated network bandwidth between proxy and storage nodes should be roughly the same as the link bandwidth used by cloud provider to connect to the proxies. Generally, this rule can be modeled as: $bw_{proxies} = storageNodes * bw_{storageNode}$.

R6 A faster network cannot effectively improve QPS for small-object intensive workloads. For tenants who do not impose strict SLO requirements, the workload, if dominated with small objects, may be better served using a combination of low-bandwidth network (i.e., 1 Gbps NICs) with high-bandwidth storage devices (e.g., SSD delivering decent random and sequential I/O performance). This low-cost heterogeneous resource combination can effectively meet tenants' requirement while improving data center cost efficiency.

R7 For large-object intensive workloads, we have to collectively consider the network bandwidth limits and the storage configuration. Given a certain network limit and SLA, a combination of slow and fast storage devices (e.g., HDD+SSD) may be able to serve the application needs in a resource efficient manner.

4.3 MOS Framework

We design MOS based on the *rules-of-thumb* developed in §4.2, with the goal to address workload and datacenter resource heterogeneity. Instead of a conventional monolithic storage architecture, where all tenants/workloads share the storage resources using same or static configurations, MOS uses multiple object stores, which we refer to as microstores. To service changing workloads, MOS performs dynamic resource partitioning and provisioning, allowing each microstore within an object storage setup to run as a fully-functional object store unit. As pointed in our earlier motivational study, although the monolithic approach is simple to implement and configure, it is not necessarily resource efficient and can lead to wastage of CPU etc. resources.

Figure 4.6 shows the architecture of MOS, which consists of two layers: (1) **Microstores:** consists of multiple instances of object stores, a *microstore*, which is allocated a subset of proxy nodes and storage nodes that matches the requirements of application the microstore will support. (2) **MOS substrate:** consists of a resource manager that monitors the load on each microstore using a workload monitor and automatically reconfigures the resources assigned to the microstore to cope with workload shifts.

Microstores. A typical MOS deployment configures multiple instances of object store. Each instance comprises a subset of nodes in the cluster, and constitutes a unique microstore. Each microstore consists of two types of nodes: proxy nodes and storage nodes (each statically equipped with *hardware resources* such as CPUs, network interface cards (NICs) and storage devices, etc.). The proxy nodes are responsible for directing queries and managing the metadata for key space and replicas, etc. Whereas, the storage nodes store the object data in their local file systems and replicate the objects at multiple storage nodes for fault tolerance. The number of microstores configured in a deployment of MOS is specified by the service provider and will depend on the kinds of workloads that need to be supported. At any given time during MOS's operation, the amount of resources allocated to each microstore is determined by its resource utilization. Such elasticity guarantees resource efficiency while delivering the needed performance.

MOS substrate. The MOS substrate layer consists of a central *Resource Manager* and the same number of *Workload Monitors* as the number of microstores. The main function of this layer is to perform online performance and resource utilization monitoring. The

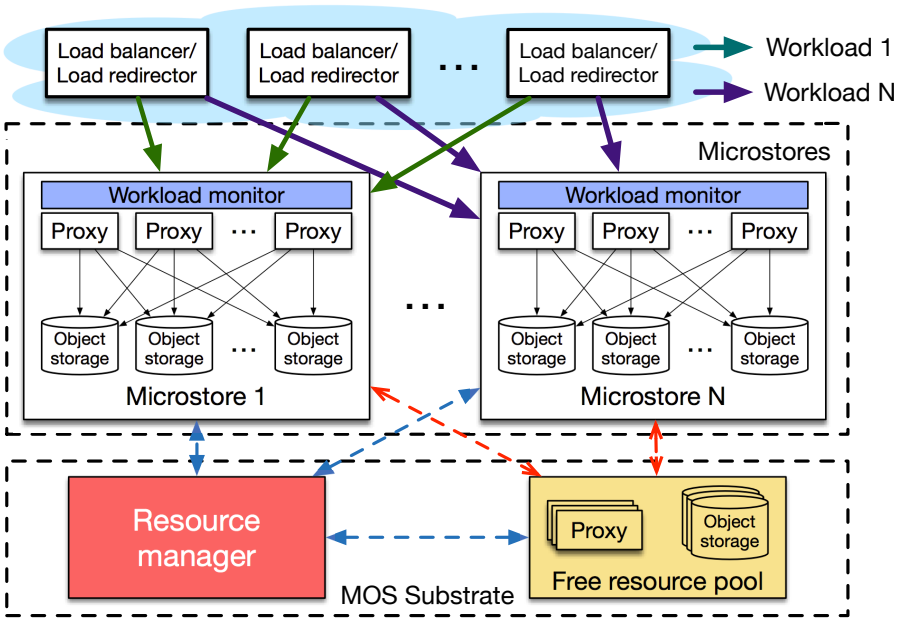


Figure 4.6: Overview of MOS architecture.

workload monitors are used to gather statistics of each running microstore, which are polled periodically by the resource manager to decide when and if a reconfiguration is warranted. The workload monitors also execute, as needed, online microstore reconfiguration commands issued by the Resource manager to meet tenants' demands. The resource manager manages the available pool of heterogeneous resources. It makes decisions about when and how to add or redistribute resources for the reconfiguration of a microstore in order to respond to workload shifts in the observed throughput and latency. We design the MOS framework in a modular fashion, where the core resource provisioning mechanism is configurable depending on tenants' needs.

4.4 Basic MOS

We first design a basic version of MOS. The goal of the basic version is to provide a coarse-grained dynamic resource management mechanism that can help achieve good resource utilization without worrying about SLA enforcement. The core of the basic MOS uses a greedy heuristic to perform resource provisioning based on online workload changes.

Basic MOS Algorithm. Algorithm 1 takes as input *microstores*, a vector of all microstores storing statistics such as hardware configuration, current load being served, and the resource utilization, e.g., CPU and network bandwidth utilization. Initially, the algorithm

Algorithm 1: MOS Resource Provisioning Algorithm.

Input: $microstores$: Microstore array, $free\_pool$: free resource pool, $util_{low}$: low utilization threshold, $util_{high}$: high utilization threshold, $epoch$: configurable monitoring interval

```
begin
    microstores.hw ← init(free_pool)
    while true do
        foreach ms in microstores do
                                                    // periodically collect monitoring stats
            if util_low ≤ util(ms.hw) ≤ util_high then
                ms.firstTime ← true
                continue
            else
                if ms.firstTime then
                    ms.firstTime ← false
                    ms.toChange ← 1
                else
                    ms.toChange ← ms.toChange * 2
                if util(ms.hw) > util_high then
                                                    // to add in more resources
                    ms.hw ← ms.hw + ms.toChange
                    /* allocate resource from free resource pool          */
                    alloc(free_pool, ms, ms.toChange)
                else if util(ms.hw) < util_low then
                                                    // to remove resources
                    ms.hw ← ms.hw − 1
                    /* return resource to free pool                       */
                    dealloc(free_pool, ms, 1)
        sleep(epoch)
```

allocates the same amount of resources to each microstore conservatively. It then enters into the main loop, where the resource manager periodically (with configurable *epochs*) polls each microstore. In each iteration, if the resource utilization (fetched using `util`($ms.hw$)) of one microstore lies within a pre-defined threshold range (i.e., [$util_{low}$ $util_{high}$]), the algorithm simply moves to the next microstore. If the microstore is in sub-optimal state, the algorithm decides to *quadratically* add or *linearly* remove resources. This is to ensure that the algorithm will not overshoot the de-allocation of resources, but can quickly respond to sudden workload increases.

4.5 Enhancements: MOS++

There are two limitations of the basic MOS resource provisioning algorithm: (i) it is not SLA-aware; and (ii) it lacks the ability to perform fine-grained resource management. We address these shortcomings by designing support for container based deployment for object store in MOS to create MOS$_{++}$. Containers greatly improve the flexibility for dynamic

reconfiguration. For example, by leveraging containers, MOS$_{++}$ can specify the number of CPU cores added to each microstore, support utilizing different types of storage devices in various configurations, and perform network partitioning.

Algorithm 2: MOS++ Resource Provisioning Algorithm (SLA-aware and Container-based).

Input: $free\_pool$: free resource pool, sla_{low}: low SLA threshold, sla_{high}: high SLA threshold, $util_{low}$: low utilization threshold, $util_{high}$: high utilization threshold, $util_{thresh}$: % amount of time sampled in one epoch, $epoch$: configurable monitoring interval

```
begin
    ms ← init(sla, free_pool)
    while true do
        if sla_low ≤ ms.perf ≤ sla_high then
            if util_low ≤ ms.util ≤ util_high then
                continue
            else
                // fine-tune the ms config
                if ms.util > util_high then
                    ms.cont ← ms.cont + 1
                    /* allocate resource from free resource pool */
                    alloc(free_pool, ms, 1)
                else if ms.util < util_low then
                    ms.cont ← ms.cont − 1
                    /* return resource to free pool */
                    dealloc(free_pool, ms, 1)
        else
            if sample(util_low ≤ ms.util ≤ util_high) > util_thresh then
                continue
            else if ms.perf > sla_high then
                                                                    // to remove containers
                toChange ← getContainers(ms.perf − sla_high) ms.cont ← ms.cont − toChange dealloc(free_pool, ms,
                toChange)
            else if ms.perf < sla_low then
                                                                    // to add in containers
                toChange ← getContainers(sla_low − ms.perf) ms.cont ← ms.cont + toChange alloc(free_pool, ms,
                toChange)
        sleep(epoch)
```

Specifying SLAs. We consider latency-based SLAs with constraints for small-object intensive workloads and bandwidth-based SLAs with constraints for large-object intensive workloads. Each SLA, which is associated with one object size attribute, has two parameters:

- Average request response time (average latency) for small-object intensive workloads *OR* average bandwidth for large-object intensive workloads. For example, for a workload where 90% of all requests have a size of 10 KB , the average latency must be within a particular range, i.e., sla_{low} and sla_{high}, where sla_{low} is associated with the

Notation	Description
E	Entities of the object store (proxy/object server)
cpu_{ij}^{E}	1 if CPU j is assigned to workload i, 0 otherwise
S	Disk storage type (PCIe/SATA SSD, HDD, etc.)
$disk_{ik}^{S}$	1 if Disk k is assigned to workload i, 0 otherwise
nw_i	Network bandwidth assigned to workload i

Table 4.2: Notations used in the LP model employed in MOS$_{++}$.

upper bound in terms of average latency and sla_{high} with the lower bound. Similarly, an SLA has a sla_{high} and sla_{low} associated with the upper and lower bound in terms of bandwidth, respectively, if the workload is large-object dominant, i.e., most objects with a size greater than 10 MB.

- Resource utilization: The fraction of time ($util_{thresh}$) the system resource utilization is within a specified range, e.g., $util_{low}$ and $util_{high}$.

The SLA requirement is met if *either one of the above two* parameters is true.

MOS++ Algorithm. Fine-grained resource allocation enables SLA-aware resource provisioning using Algorithm 2, which is an enhancement of Algorithm 1. In addition to the input parameters provided to the basic Algorithm 1, the enhanced algorithm takes three extra parameters: sla_{low}, low SLA threshold; sla_{high}, high SLA threshold; and $util_{thresh}$, fraction of time slots sampled during the period of one epoch.[2] In a workload, if the performance of a microstore goes above sla_{high}, resources are removed and reclaimed at per-container granularity from that microstore based on the suggested value provided by the function $getContainers(.)$. Conversely, resources are allocated and added into the microstore whose performance is observed to go below the sla_{low} value. We set a precondition of resource utilization as the second parameter for our SLA. Function $sample(.)$ periodically measures the resource utilization on a per-epoch basis. Again, the resources that need to be added are suggested by function $getContainers(.)$, which uses a linear programming (LP) optimizer to compute a near-optimal allocation plan.

Resource Provisioning Optimization. We define the amount of resources allocated to workload i as:

$$resource_i = \sum_j cpu_{ij}^{E} + \sum_l disk_{il}^{S} + nw_i \ . \tag{4.1}$$

[2]The sampling probability (%) is configurable. Each time slot is configured as 1% of the epoch. E.g., for an epoch of 5 minutes, a sampling time slot is 3 seconds.

The resource provisioning problem is modeled as an optimization problem to minimize the resources used for all workloads. Specifically, the objective is to:

$$minimize \sum_i resource_i \,, \tag{4.2}$$

$$s.t.\ sla_i^{low} \leq \text{cosperf}(resource_i) \leq sla_i^{high}, \forall i \,, \tag{4.3}$$

$$0 \leq \sum_j cpu_{ij}^E \leq 1, cpu_{ij}^E = 0 \text{ or } 1, \forall i \,, \tag{4.4}$$

$$0 \leq \sum_l disk_{il}^S \leq 1, disk_{il}^S = 0 \text{ or } 1, \forall i \,, \tag{4.5}$$

$$0 < \sum_i nw_i \leq nw_{max} \,. \tag{4.6}$$

Table 4.2 describes the notations used for representing the above model. Constraint 4.3 is used to guarantee that the SLA (the average response time for small-object workloads and the average bandwidth for large-object workloads) requirement is met. We profile the performance offline using an extensive stress test by iterating through nearly all possible resource configurations (§4.2). The estimated performance of a particular workload is estimated using function $cosperf(.)$, fed with the provisioned resources. Constraints 4.4 and 4.5 make sure that a CPU core or a storage drive can only be assigned to one workload. Similarly, Constraint 4.6 restricts the maximum network bandwidth (nw_{max}) that can be allocated to all the workloads. In the context of object stores, the CPU resources are allocated to two entities (denoted using E): proxies and object servers. While our model is general enough to cover many types of storage devices, we focus on three extant storage types (denoted using S) – PCIe/SATA SSDs, HDDs – in this work. The fairly small problem size implies that the optimization problem can be solved quickly, i.e., in seconds using CPLEX [10].

4.6 Implementation

Figure 4.7 shows the implementation details of MOS and MOS$_{++}$. We build MOS on top of the Mesos [118] framework. The Mesos resource management is driven by the resource provisioning algorithms, i.e. Algorithm 1 or Algorithm 2. MOS launches Swift directly on the physical nodes serving as Mesos slaves. For MOS$_{++}$ we extend Volt [59], a Mesos framework that can be used to launch containers on Mesos slaves.

For fine-grained resource allocation, we enforce the runtime constraints on resources while launching the Docker containers using the options provided by Docker [14]. The number of CPU cores is configurable for launching applications inside a container. Docker supports CPU core binding and disk size partitioning. For efficient disk utilization, we launch the object server containers with privileged mode, which enables the usage of devices attached to the host machine inside the container. We leverage this option to attach suitable storage

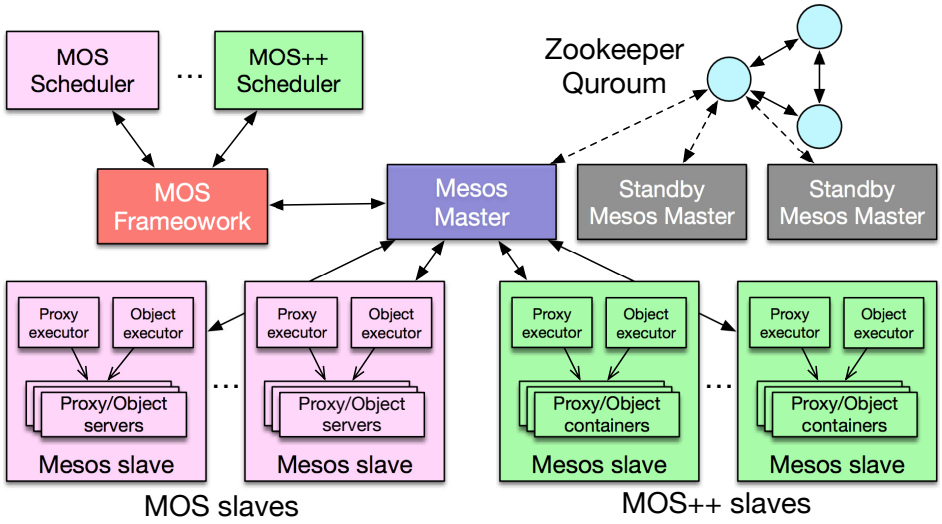


Figure 4.7: MOS modules and their interactions. Mesos provides support to launch Swift either directly on physical nodes or inside containers.

devices for launched containers. To perform network partitioning, we use control groups (`cgroups`) [7] with Linux traffic control (`tc`) [35].

Container based deployment of Swift offers several advantages such as rapid deployment, portability across machines, easy sharing, lightweight footprint and simplified maintenance. All these advantages come at a cost of negligible or "close to zero" overhead [111]. To provide high availability, the Mesos cluster is launched with 3 masters with one acting as the leader and the rest on standby. In case of a master failure, a standby leader becomes the leader as a result of the election done by Zookeeper [122].

We have implemented our own proxy executor, object executor, and Python scripts to automate online reconfiguration of Swift in a distributed setup. The proxy executor is used to support dynamic reconfiguration of proxy servers inside the containers. The object executor performs online disk capacity management using the `swift-ring-builder` tool[3], whenever the storage configuration is updated. The code to build Docker container images and the Docker images for Swift proxy and object server are publicly available on Github [16, 17].

[3]Swift lazily migrates existing data replicas to newly added disks where new data can be instantly written and persisted. We leverage this feature to (i) control the I/O performance on new data that is written in newly added disks, and (ii) amortize the data migration cost.

4.7 Evaluation

We present the evaluation of MOS$_{++}$ using both a prototype implementation and simulations. We first use the prototype to evaluate a number of object store setups under multi-tenancy in both static and dynamic workloads. This is followed by a simulation study of a large-scale system to compare MOS$_{++}$ and MOS.

4.7.1 Prototype Evaluation

Experimental Methodology. We evaluate MOS$_{++}$ using a 128-core local testbed. The testbed is connected using a 10 Gbps switch, with a maximum bandwidth of 40 Gbps. We emulate a two-tenant (client) environment, i.e., we run COSBench on two separate machines within the same subnet. We use `WorkloadA` (small-object read-intensive workload) and `WorkloadC` (large-object read-intensive workload) for this purpose. We compare MOS$_{++}$ against two different object store setups – `Default`, where we use off-the-shelf monolithic configuration of Swift, and `Static`, where we statically configure two micro object stores designated for two tenants based on the rules-of-thumb of §4.2. The static approach is more advanced than the default segmentation policies [57] and serves as another point of comparison for our approach. Note that we focus on MOS$_{++}$ for our prototype evaluation as it also encompass the basic design of MOS.

The `Default` setup is launched directly on the physical machines. `Static`, like MOS$_{++}$, is launched inside containers. For `Static` setup, we tried several different overall configurations and selected the best one. Specifically, 75% CPU cores, 30% of NW bandwidth, and 100% PCIe SSD with 30% SATA SSD are assigned to `WorkloadA`. Accordingly, 25% CPU cores, 70% NW bandwidth, and 70% SATA SSD are assigned to `WorkloadC`. MOS$_{++}$ starts initially with the same configuration as `Static` throughout our evaluation. Regarding runtime parameters, we set sla_{low} to be proportional to the workloads' load and sla_{high} 2× of sla_{low}. We set $util_{low}$ 65% and $util_{high}$ 85%. We set *epoch* to be 3 minutes and $util_{thresh}$ as 80%.

Performance Evaluation. In our first set of experiments, we evaluate MOS$_{++}$'s ability to handle heterogeneous varying workloads. We vary the COSBench processes from 2 to 1024 for `WorkloadA` to increase the throughput, while we decrease `WorkloadC`'s load by varying the COSBench processes from 32 to 2. Figure 4.8 plots the overall performance of the two studied workloads in terms of both throughput (QPS) and bandwidth (MB/s). `Default` achieves significantly higher bandwidth compared to `Static` when `WorkloadC` dominates (the far left part on X-axis dimension). This is because the large-object workload consumes most of the network bandwidth to transfer packets containing payload for large objects. Guided by our rules-of-thumb, `Static`'s statically provisioned micro store setup is able to balance the performance of both workloads to some extent. Hence, as `WorkloadA` gradually increases and eventually dominates, `Static` outperforms `Default` by as much as 2×. By leveraging

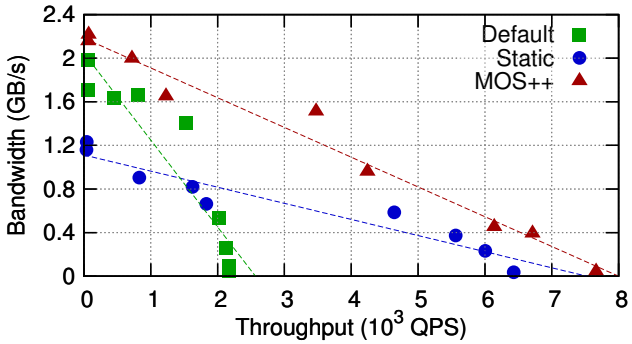


Figure 4.8: Overall throughput vs. bandwidth observed under different setups. Dotted lines are generated using linear regression, indicating the linear relationship between the overall throughput and bandwidth.

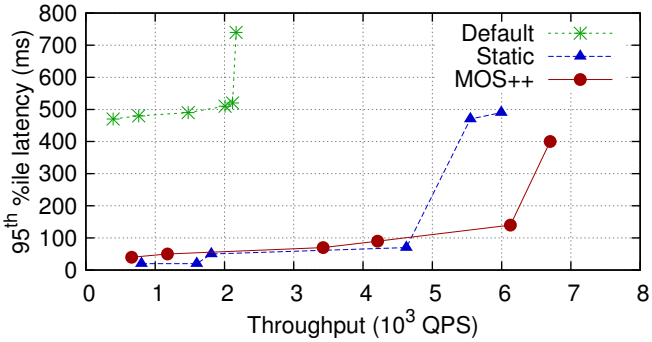


Figure 4.9: Throughput vs. 95^{th} percentile latency under `WorkloadA`.

workload-aware elasticity support, MOS$_{++}$ combines the "best of both worlds", hence we see 10.4–89.6% improvement in overall throughput and 7.6–79.8% improvement in overall bandwidth, compared to both `Default` and `Static`. Thus, MOS$_{++}$ is able to improve the overall performance for the two tenants with workloads exhibiting dramatically different characteristics.

Figure 4.9 depicts the 95^{th} percentile read tail latency and throughput tradeoffs observed for `WorkloadA`. For `WorkloadA`, `Default` performs the worst and lies in the upper-left corner of the scatter chart. `Static` achieves comparatively similar performance with MOS$_{++}$ as `WorkloadA` starts to increase. By adapting to the increasing load and adding more CPU power for `WorkloadA`, MOS$_{++}$ eventually outperforms `Static` at peak loads (the right-most two data points) by up to 11.7% in throughput and up to 70.2% in tail latency. Figure 4.10 shows a similar trend under `WorkloadC`. Under the large-object dominant workload, `Static`

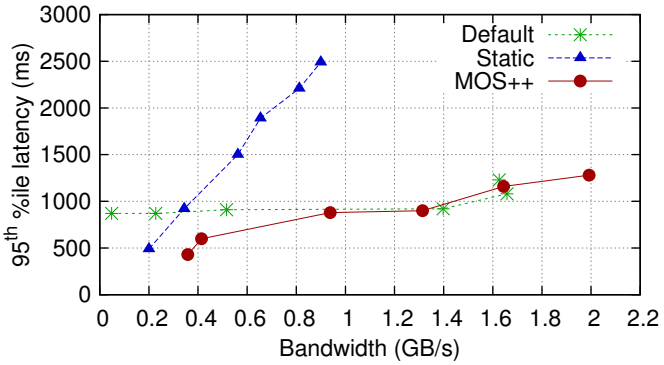


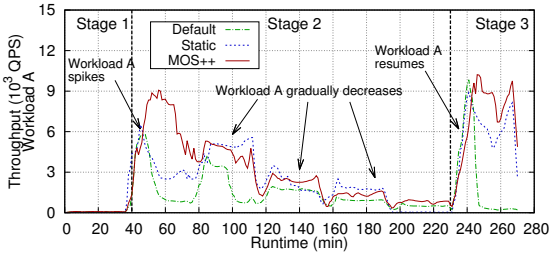
Figure 4.10: Bandwidth vs. 95^{th} percentile latency under `WorkloadC`. Average latency results show similar trend.

is bottlenecked by its statically allocated network resource and hence limits the bandwidth for `WorkloadC`. Accordingly, we observe up to 79% improvement in bandwidth and up to 50.6% reduction in tail latency, compared to `Default`. Thus, MOS$_{++}$ is able to improve the performance for both tenants, and effectively remove the performance bottleneck observed in `Default` and `Static` setups.

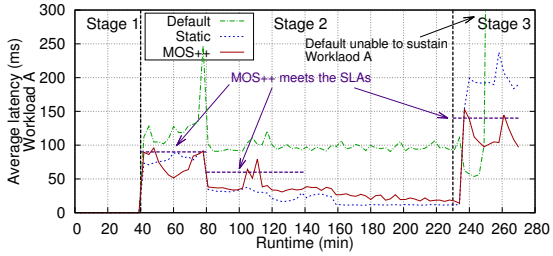
Adaptivity and Efficiency Evaluation. In our next experiment, we evaluate the adaptivity and resource efficiency of MOS$_{++}$. We use a dynamically changing heterogeneous workload, which is issued concurrently by two tenants – `WorkloadA` and `WorkloadC`. The workload is generated by COSBench and is composed of three stages. In `Stage 1`, the workload is dominated by large objects (`WorkloadC`). At around 40 min, `Stage 2` begins with abrupt change in workload characteristics as the small-obj dominant workload (`WorkloadA`) instantaneously spikes and then gradually shifts down. This lasts for until around 200 min. Finally, at 230 min, in `Stage 3` there is another abrupt change as `WorkloadA` once again increases and dominates. With these three stages concatenated together we capture the behavior of MOS$_{++}$ both under abrupt as well as gradual change in a dynamically changing workload in a multi-tenant environment. Figure 4.11(a) and Figure 4.11(b) plot the average throughput change and the average read latency change[4] of `WorkloadA`, respectively. Figure 4.11(c) plots the average bandwidth change of `WorkloadC`. Besides, Figure 4.12 depicts the breakdown of a variety of resources used in our tests.

As `Stage 1` begins, MOS$_{++}$ achieves the same performance as `Static`. This is because MOS$_{++}$ starts off with the same setup as `Static`. After around 5 min, MOS$_{++}$ slightly outperforms `Static` by 12%, because MOS$_{++}$ is SLA-aware and gradually reduces the network bandwidth originally allocated to `WorkloadA` and its performance reaches the highest after 5 min. Though `Static` achieves comparatively similar performance, it is not able to satisfy

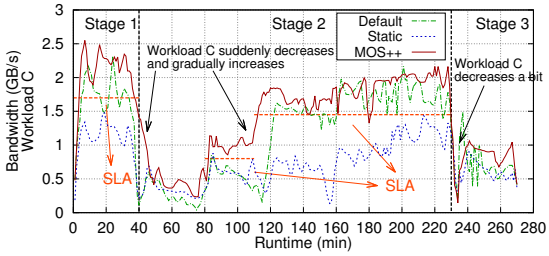
[4]We observed a similar trend for write latency.



(a) Throughput under `WorkloadA`.



(b) Average latency under `WorkloadA`.



(c) Bandwidth under `WorkloadC`.

Figure 4.11: Performance of MOS$_{++}$ and the baselines under a dynamically changing heterogeneous workload. Purple dashed lines represent tenant-defined SLAs in terms of average latency (sla_{low}) for `WorkloadA`. Orange dashed lines represent tenant-defined SLAs in terms of bandwidth (sla_{low}) for `WorkloadC`.

the SLA at 100% since its allotted network bandwidth is quickly saturated. In contrast, as observed in Figure 4.12, `WorkloadC` eventually uses up to 95% of all the available network resources (**R3** and **R5** in §4.2). Accordingly, MOS$_{++}$ detects nothing significant in `WorkloadA` and decides to free up the CPU resource originally allocated to `WorkloadA` (recall that MOS$_{++}$ and MOS starts with the same configuration). This clearly demonstrates that, rather than reserving resources statically, MOS$_{++}$ elastically reprovisions the resources to achieve better resource efficiency while meeting the SLA.

`Stage 2` instantaneously begins at around 40 min and quickly spikes. At this time, MOS$_{++}$ immediately start to react. Driven by the increasing `WorkloadA` and the specified SLA (90 ms average latency), MOS$_{++}$ provisions proxies and object stores with more CPU cores. This can be observed in Figure 4.12 (**R2** and **R4**). Note in Figure 4.11(a), MOS$_{++}$ improves the throughput of `WorkloadA` by up to 2.1× at around 60 min. As `WorkloadC`'s load decreases as `Stage 2` begins, MOS$_{++}$ reduces the CPU cores allocated to `WorkloadC` and reassigns them to `WorkloadA`. Correspondingly, 58% of the network bandwidth allocated is reclaimed and marked as free and 21% is reallocated to `WorkloadA` (**R6**). `WorkloadA` spikes at around 60 min and gradually decreases while `WorkloadC` slowly increases. Accordingly, MOS$_{++}$ adapts by reclaiming the extra CPU resources that are not needed for the current load-

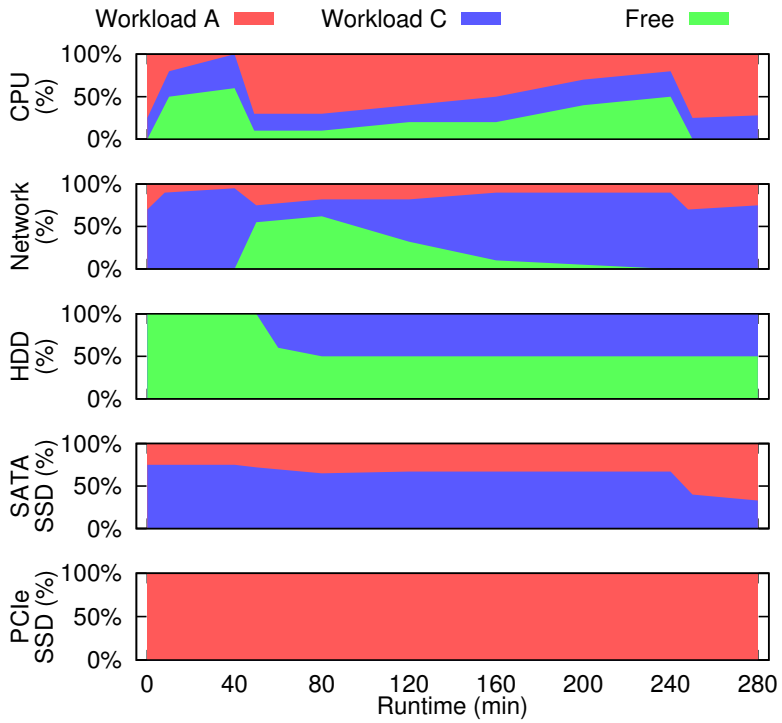


Figure 4.12: Resource allocation breakdown under dynamically changing heterogeneous workload.

/SLA for `WorkloadA`, and grabs back the network resource, which gets eventually saturated by `WorkloadC` (Figure 4.11(c)). Note that MOS$_{++}$ allocates 50% of the HDD capacity to `WorkloadC` at 80 min, since the optimizer detects that a combination of HDD+SSD yields the most economical setup provided the given SLA (**R7**). As shown in Figure 4.11(a), MOS$_{++}$ maintains the highest performance for both tenants while being resource efficient, whereas both `Default` and `Static` fail to do so, which further demonstrates the superiority of MOS$_{++}$.

`Stage 3` begins at around 230 min with a sudden spike of `WorkloadA` and about half drop in `WorkloadC`. `Default` is unable to sustain `WorkloadA`, because the network bandwidth is mostly used up by `WorkloadC`. Again, MOS$_{++}$ maintains the best performance for both tenants featuring the SLA-awareness and workload adaptivity.

4.7.2 Comparison of MOS and MOS++

In our next set of experiments, we use a simulation study to compare MOS and MOS$_{++}$ for a large-scale setup.[5]

[5]For comparison of MOS with `Static` and the monolithic `Default` setup please refer to our short work [70].

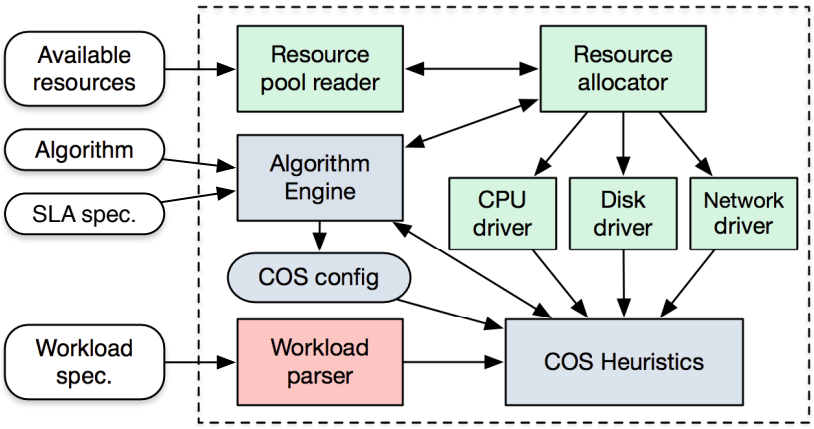


Figure 4.13: COSPerf architecture.

COSPerf Simulator. We design and implement a cloud object store simulator (COSPerf) based on the rules-of-thumb discussed in §4.2. Figure 4.13 depicts the architecture of COSPerf. A resource parser takes as input available resources, and sort them based on capability/capacity. The resources from these pools are then fetched by the resource allocator for launching resource drivers including a CPU driver, a disk driver, and a network driver. Resource allocator is driven by the algorithm engine, which is used to simulate the resource provisioning algorithm that is provided as a configurable parameter to COSPerf. The algorithm engine interacts with the resource allocator to keep track of the available resources and generates an object store configuration file. The algorithm engine also monitors the simulated performance of the launched microstores.

A cloud object store (COS) heuristics module takes workloads as input and predicts performance under a set of specified software/hardware configurations. The dynamic shifts in simulated performance due to online reconfiguration and workload changes are modeled and calculated by the COS heuristics module, which is built based on the extensive analysis and profiling of a real object store under various scenarios (§4.2). Thousands of experiments (each running for 15 minutes) were executed under varying load/configuration to further fine-tune our COS heuristics model.

Simulation Methodology. For simulation, we assume a pool of 50 server machines with diversified (heterogeneous) hardware configurations, including CPU, network, and storage devices. Specifically, we use the following pool of resources: (i) 3 32-core machines, 4 16-core, 31 8-core machines, and 12 4-core machines; (ii) 18 10 Gbps and 32 1 Gbps NICs; and (iii) HDD to SSD ratio was 70% to 30%. The workloads run for about 14 hours and we divide the workloads in four major stages. For analysis purposes, we focus on throughput in terms of QPS for small object workloads and bandwidth for large objects.

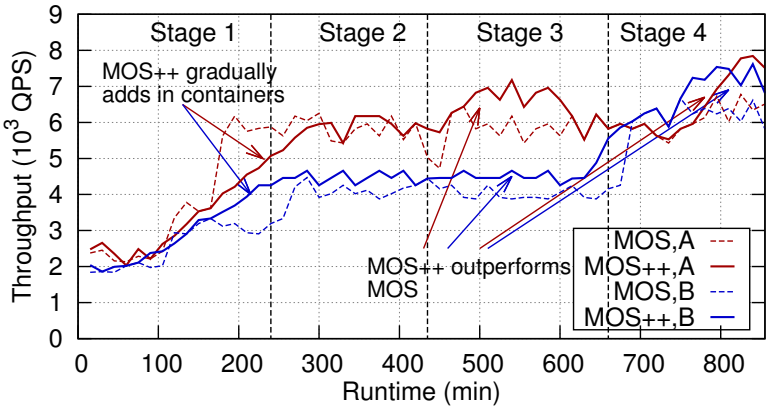


Figure 4.14: Throughput of MOS Vs. MOS$_{++}$ under `WorkloadA` and `WorkloadB`.

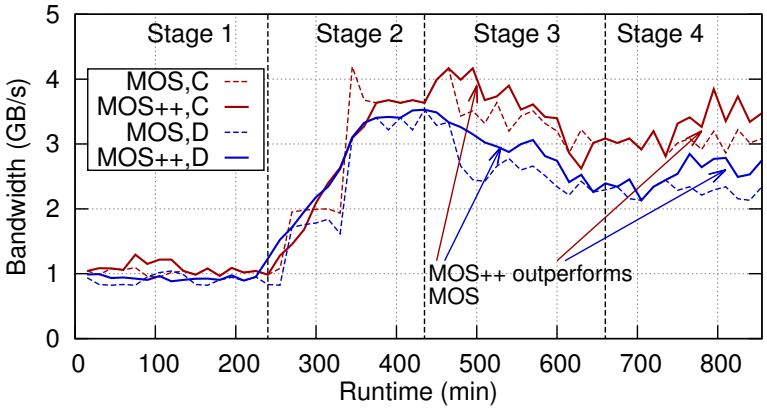


Figure 4.15: Bandwidth of MOS Vs. MOS$_{++}$ under `WorkloadC` and `WorkloadD`.

Simulation Evaluation. Figure 4.14 and Figure 4.15 show how MOS and MOS$_{++}$ behave under dynamically-changing `WorkloadA-D`. The goal of our evaluation is to test under both abrupt and gradual change in a workload under multi-tenancy, which are emulated by our workloads. In `Stage 1`, as the load of small-object workloads (i.e., `WorkloadA` and `WorkloadB`) increases (as shown in Figure 4.14), MOS uses resources from the free resource pool to keep the resource utilization under control until the utilization stabilizes and falls back in the $[util_{low}, util_{high}]$ range. MOS$_{++}$ also achieves the same goal but by allocating resources at a finer granularity. As a result, we do not see a sudden spike in QPS. Instead the transition is smoother, which allows both `WorkloadA` as well as `WorkloadB` to almost linearly increase QPS. In addition to maintaining resource utilization within the acceptable range, MOS$_{++}$ also makes sure that SLA requirements are met.

As `Stage 2` begins, the load of large-object workloads (i.e., `WorkloadC` and `WorkloadD`) in-

Workload	MOS	MOS++
A	4444 QPS	4994 QPS
B	3828 QPS	4429 QPS
C	2 GB/s	2.3 GB/s
D	1.6 GB/s	1.9 GB/s

Table 4.3: Average performance summary.

creases. As shown in figure 4.15, both MOS and MOS$_{++}$ start adding resources to accommodate the increasing demand on network bandwidth. Again, MOS$_{++}$ keeps fine-grained track of the resources at per-container basis and just allocates the right amount of resources to meet the tenant requirements while maximizing the resource efficiency. Starting around 310 min, however, MOS ends up adding more resources than needed to microstore C and D. Though MOS is able to lower down the network utilization from 95% to 85%, it causes a spike in performance.

From the beginning of `Stage 3` up until around 500 min, both MOS and MOS$_{++}$ end up utilizing all the resources due to the increasing demands from all tenants. As shown in Figure 4.14 and Figure 4.15, MOS$_{++}$ outperforms MOS by up to 25% for `WorkloadA`, and up to 31% for `WorkloadD`. This is due to the following design choices we make in MOS$_{++}$: (i) MOS$_{++}$ allocates resources at the container granularity; and (ii) MOS$_{++}$'s optimizer generates a better resource provisioning plan that yields higher performance by exploiting all the available resources. After 500 min, the load decreases for `WorkloadC` and `WorkloadD`. As a result, MOS and MOS$_{++}$ reclaim resources from microstore C and D to the pool of free resources. Finally, in `Stage 4`, the load further increases for `WorkloadA` and `WorkloadB`. As a result, MOS and MOS$_{++}$ utilize the resources freed up in `Stage 3` from `WorkloadC` and `WorkloadD`. At this point, both MOS and MOS$_{++}$ quickly detects performance improvement opportunity for `WorkloadB` as the throughput of `WorkloadB` is still at a low level, while more resources are also added into `WorkloadA` with the goal to maintain the CPU utilization within the "sweet" range. Hence, tenants will not see performance lost as the workload shifts. MOS$_{++}$ further improves MOS's performance on `WorkloadA` and `WorkloadB` by up to 33% and 26%, respectively. This, again, demonstrates the superiority of MOS$_{++}$ in effectively utilizing the limited resources for maximizing performance improvement and meeting tenants' SLAs.

Table 4.3 presents a summary of the average performance by combining all stages for `WorkloadA-D`. The results show that for small-object workloads, MOS$_{++}$ achieves 12.4% better performance for `WorkloadA`, and 15.7% better performance for `WorkloadB`, compared to MOS. Similarly, average performance of MOS$_{++}$ for large-object workloads is 15% higher for `WorkloadC` and 18.8% better for `WorkloadD` than that of MOS.

4.8 Discussion

There are two limitations that are not fully addressed in our current implementation. First, although MOS supports multi-tenancy and heterogeneous workload separation, we limit the number of microstores to be launched based on workload characteristics (i.e., object sizes) to reduce the implementation complexity and reconfiguration overhead. Consequently, it limits the kinds of different workloads the system can effectively handle. Should a workload change its inherent characteristics, e.g., the object size distribution changes dramatically, and no longer fit well with any provisioned microstores, the system may end up doing reconfiguration thrashing. This in turn will lead to reduced performance. A possible solution is to perform online workload analysis and profiling at the load balancer/redirector side, and using the information to compute an optimal number of microstores and perform workload-to-microstore mapping on the fly. Such a dynamic detect-and-map system is part of our future work. Second, although MOS$_{++}$ is able to meet the SLAs by leveraging offline workload profiling and online optimization, it does not currently consider the profit, i.e., revenue, for the service provider and tenant utility, i.e., $perf/\$$, while provisioning the microstores. A feasible yet simple cloud-profit-aware solution can be to enhance our optimizer by incorporating the cloud pricing model and monetary profit. This aspect is orthogonal to our work, but can be easily incorporated into the design if needed.

4.9 Summary

In this work, we have presented an experimental analysis of cloud object store, and proposed a set of rules-of-thumb based on the study. The rules provide practical guidelines for service administrators and online resource managers to better tune object store performance to application needs. The resulting system, MOS, outperforms extant object stores in multi-tenant environments. Furthermore, we build MOS$_{++}$ to enhance MOS by leveraging containers for fine-grained resource management and higher resource efficiency. Our experimentation reveals that it is possible to exploit the inherent heterogeneity within modern datacenters to better serve heterogeneous workloads across multiple tenants. Evaluation with our prototype implementation shows that MOS$_{++}$ improves performance by up to 89.6% and 79.8% compared to the default monolithic and statically configured object store setup, respectively. We have implemented COSPerf, a cloud object store simulator, to further verify the design choices of MOS$_{++}$. Results show that, by utilizing the same set of resources, MOS$_{++}$ achieves up to 18.8% performance improvement compared to the basic MOS.

Chapter 5

Functional Partitioning of Object Stores

5.1 Introduction

Distributed Object-Based Stores (DOBS) such as key-value (KV) stores and message queues play an increasingly critical role in supporting today's large-scale web services. Relatively simple data schemas and indexing enable DOBS to achieve high performance and scalability, as well as quick request processing. Consequently, more and more specialized DOBS for different use cases are being designed and developed. For example, Amazon employs Dynamo [105] for online shopping service with eventual consistency guarantee; Expedia is powered by the strongly consistent MongoDB [40] for serving the flight and trip reservation services; and LinkedIn's central data pipeline is powered by the Kafka [32] distributed messaging system.

While developers[1] are free to choose existing DOBS or implement a new system from scratch to meet their requirements, they are faced with common challenges, which this work aims to address.

First, designing and implementing a new highly available, resilient, and efficient distributed storage system is not an easy task. A key observation is that although every DOBS adopts different designs to meet various requirements, they still share a set of features such as handling cluster topology, guaranteeing consistency, and supporting replication and fault tolerance. Table 5.1 shows the common features in modern DOBS. Some support master-slave (MS) topology, while others support active-active (AA)[2]. Similarly, some DOBS guarantee strong consistency (SC), while others provide eventual consistency (EC). A subset support both replication and fault tolerance. Based on this observation, we foresee significant po-

[1] We use 'developers' to mean 'DOBS' developers throughout.

[2] AA is also called multi-master in database terminology.

Systems	TY	TO	C	R	FR	OB
HyperDex [109]	KV	MS	SC	✓	✓	-
FAWN [66]	KV	MS	SC	✓	✗	-
MongoDB [40]	KV	MS	SC	✓	✓	-
PNUTS [102]	KV	MS	EC	✓	✓	-
Memcached [37]	KV	✗	✗	✗	✗	-
etcd [22]	KV	AA	SC	✓	✓	-
Cassandra [131]	KV	AA	EC	✓	✓	-
Dynamo [105]	KV	AA	EC	✓	✓	-
Redis [45]	DS	MS	EC	✓	✗	-
TAPIR [168]	LIB	AA	SC	✓	✗	-
Kafka [32]	MQ	MS	EC	✓	✓	-
Dynomite [19]	FW	AA	EC	✓	✗	✓
Ceph [165]	OBJ	MS	SC	✓	✓	-
S3 [52]	OBJ	AA	EC	✓	✓	-
Swift [56]	OBJ	MS	EC	✓	✓	-

Table 5.1: A (partial) list of DOBS representing the state of the art. `TY`: Type (where '`KV`' stands for key-value store, '`DS`' data structure store, '`LIB`' library, '`FW`' framework, '`OBJ`' object store, 'MQ' message queue); `TO`: Topology; `C`: Consistency; `R`: Does replication; `FR`: Has failover & automatic recovery; `OB`: Backend data store oblivious; `-`: Standalone & monolithic DOBS (no multiple data stores); `MS`: Master-slave topology; `AA`: Active-active topology; `SC`: Strong consistency; `EC`: Eventual consistency.

tential benefits of enabling modular and compositional development of DOBS, e.g., starting from integrating pre-built modules for the needed common features and extending them to satisfy specialized needs.

Second, distributed systems are bug-prone and notoriously difficult to implement [87, 106]. Researchers have found that different distributed systems share common bug patterns [116, 134, 135]. For example, both Cassandra and ZooKeeper [121] suffer similar protocol bugs (e.g., *Cassandra-6023*, and *ZooKeeper-335*) in Paxos coordination. Moreover, developers of HBase [26] and Redis [43] made similar mistakes in the failover module (e.g., *HBase-3380* and *Redis-1381*). With a versatile framework as envisioned by this work, more efficient concentrated testing and debugging becomes possible.

Last but not least, continuous change in business, i.e., application needs, will drive changes in storage requirements. Existing DOBS designs entail deciding many design points at development time, such as cluster topology and consistency model. This can potentially lead to issues such as vendor lock-in, when, say a business's needs change from requiring eventual consistency to strong consistency and accommodating such a need requires a disruptive and costly migration. A more flexible and versatile solution for designing DOBS is needed.

This work presents ClusterOn, a modular and compositional development platform that addresses the above issues and eases the overhead of designing and implementing DOBS. ClusterOn is based on two key insights: (1) DOBS design composes a *control plane* respon-

sible for distributed management, e.g., replication, consistency, and topology, and a *data plane* that actually stores data; and (2) different DOBS support a common set of management functionalities in the control plane, and such common features can be modularized and reused for building different/new DOBS. Decoupling the control and data planes in software defined networking [146] has revolutionized the way networking software is developed and managed. ClusterOn envisions a similar revolution for DOBS development.

At a high level, ClusterOn provides a set of management units, referred to as *controlets*, that run on all participating nodes and provide common distributed store management functionalities, e.g., MS and AA topologies, SC and EC models, replication and fault tolerance mechanisms. Users provide what we refer to as *datalet*, which is a non-distributed version of the data store to be implemented and provide core data store functionality. They are relieved from managing the *messy plumbings* of a distributed system, which is offloaded to ClusterOn. ClusterOn spawns controlets on participating nodes, which in turn launches one or more instances of a datalet to create a DOBS. Similar to the extant services, such as libMemcached [33], and Redis clients [44], ClusterOn also provides a library that can be linked with the users' applications to utilize the DOBS services.

Moreover, ClusterOn can be extended for supporting new DOBS, e.g., a distributed message queue based on a basic data store, or KV stores with hybrid topologies. Thus, in essence, ClusterOn leverages the reusability principles [130] to simplify the task of constructing systems with new designs.

5.2 ClusterOn Design

In this section, we describe the design of ClusterOn and how it provides compatibility, versatility, modularity, and high performance for supporting DOBS.

5.2.1 ClusterOn Architecture

Figure 5.1 shows the overall architecture of ClusterOn comprising four modules: datalet, controlet, coordinator, and client library. Figure 5.1(a) shows the logical layout of ClusterOn, with the datalets making up the data plane, and the rest of the modules making up the control plane. Figure 5.1(b) shows the physical placement of the modules in a distributed setup.

Datalet is supplied by the user, and is mainly responsible for actually storing data within a single node. It should provide the basic I/O interfaces for the DOBS to be implemented. We refer to this interface as the datalet API, e.g., a datalet API can consist of `Put` and `Get`. For example, a user can develop a simplest form of in-memory hash table which supports the datalet API and provides the resulting *datalet* to ClusterOn for building a DOBS.

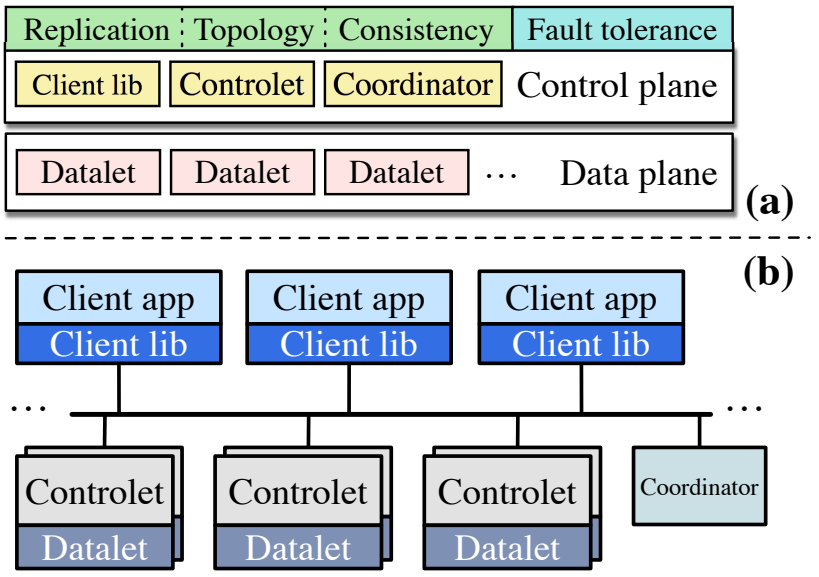


Figure 5.1: ClusterOn's logical (a) and physical (b) architecture.

Controlet is supplied by ClusterOn and provides a datalet with distributed management services to realize and enable the DOBS associated with the datalet. The controlet processes client requests and routes the requests to the associated entities: e.g., to datalet for storing data. ClusterOn allows an arbitrary mapping between a controlet and a datalet. A controlet may handle N ($\geq$ 1) instances of datalets, depending on the processing capacity of the controlet and its datalets, and can leverage physical resource (datacenter) heterogeneity [107, 145] for better overall utilization. For instance, a controlet running on a well-endowed node may manage more datalet nodes than a controlet running on a low-capacity node. For simplicity, we use one-to-one controlet–datalet mapping in the rest of the work. Moreover, we also provide support for advanced users to easily design new controlets as needed for realizing a DOBS that may require specialized handling in the controlet.

Coordinator is a special form of a controlet provided by ClusterOn. It supports the following three functions. (1) It maintains the metadata regarding the whole cluster topology and provides a metadata query service by running a metadata server. (2) It tracks the liveness of the cluster by exchanging periodic heartbeat messages the controlets. (3) It coordinates failover in case of a node failure. The coordinator can run on separate node or alongside other controlets. Moreover, similar to designing specialized controlets, advanced users also have the option to design customized coordinators if needed.

Client library is provided by ClusterOn and used by the client applications to utilize the DOBS services created by ClusterOn. The library provides a flexible means for mapping data to controlets. The client application use the library interface to consult with the coordinator and fetch data partitioning and mapping information, which is then used to route requests

to appropriate controlets. ClusterOn allows different DOBS to choose their own partitioning techniques such as consistent hashing and range-based partitioning.

5.2.2 Data Plane

A collection of datalets running on different distributed nodes form the data plane for ClusterOn. However, datalets are completely unaware of each other and the distributed setup, which enables us to support a modular DOBS design and development.

Data plane development For the ease of development, ClusterOn provides datalet templates based on commonly used data structures. Currently, we support a hash-table-based *vHT*, a log-based *vLog*, and a tree-based *vMT* template. Furthermore, ClusterOn also allows using existing datalets (or single node instances of a store), e.g., by realizing a DOBS atop SSDB and Redis. Our templates also realize network messaging and an event framework. Thus, developers can use the templates to build a variety of powerful and scalable DOBS by writing a few hundred lines of code.

5.2.3 Control Plane

As mentioned earlier, ClusterOn provides a control that offers management of many kinds of datalets. However, to support advanced users and new kinds of DOBS, ClusterOn also exposes an *event-driven* programming model for controlet development. For each event (e.g., `Put` request, timeout, etc.), ClusterOn allows developers to define event handlers to instruct how the controlet should process the event in a flexible manner to enable versatile distributed management services in the control plane.

Control plane development Developers can design controlet from scratch. However, to facilitate reusability and modularity in DOBS development, ClusterOn identifies four core components for distributed management, and provides pre-built controlet event handlers that support common design options in existing DOBS. The choice is based on our comprehensive study of Table 5.1 that revealed three key observations: (1) cluster topology, consistency model, replication, and fault tolerance generally define distributed features of DOBS; (2) for topology, MS and AA are common; and (3) for consistency model, SC and EC are popular.

Furthermore, ClusterOn provides a set of libraries with common features that can be used for controlet development. One example is providing concurrency control for Distributed Lock Manager (DLM). In this context, among many techniques such as TAPIR-based lock [168], the Redlock algorithm [48] matches well with ClusterOn architecture, as Redlock isolates control logic from data management that is backed with Redis. If desired, one can build a stand-alone DLM server using Redis as a datalet. However, to avoid an extra hop to the locking layer, ClusterOn provides the Redlock-based *DLM library*, which exposes `Lock`/`Unlock` interfaces. This would allow a more efficient realization of a DLM server.

Similarly, it is also possible to design a new coordinator as a special form of controlet from scratch. Nonetheless, because it is widely used across many DOBS, ClusterOn includes the coordinator as a default module in the control plane and exposes event interface to developers if specialized coordinatoion is desired. For example, developers can express fault tolerance strategy, metadata management, etc., for their new coordinator.

Control plane configuration To configure the system, each controlet takes as input (1) a JSON configuration file that specifies the basic system deployment parameters such as topology, consistency model, the number of replicas, and coordinator address; and (2) a datalet host file containing the list of datalets to be managed. ClusterOn loads the runtime configuration information at the coordinator, which serves as the query point for the client library and controlets to periodically retrieve configuration updates. Any changes in configuration at runtime (e.g., topology change) results in the associated controlet updating the coordinator (by switching event handlers) accordingly.

Replication and Topology ClusterOn maintains a logical relationship among individual datalets that store the replicated data whereas controlets along with client library decide the control flow to answer *when*, and *where* to send *what* messages. Generally, most distributed storage applications can be divided in two types of topology models, namely, (1) master-slave (MS), and (2) active-active (AA), or some combination thereof. In MS, ClusterOn specifies one controlet–datalet pair as master and the rest of $N-1$ replicas as slaves (N is the configurable number of replicas). Writes go to the master replica and reads go to one or more slave replicas. In AA, all replicas serve both reads and writes concurrently. The current ClusterOn prototype supports both the MS and AA topologies, and can be extended to support other variants.

Consistency To enforce a certain consistency model, extra logic need to be added in a collaborative fashion within both client-side mapping and controlet message routing. This logic ensures the timeliness of each action to satisfy the *when* constraint. For example, to support strong consistency, a `Get(key)` returns the value of the last preceding `Put(key)` performed by and acknowledged to any client. To support eventual consistency, a `Get(key)` returns the value written by any `Put(key)`, i.e., any version of the object with the given key. Clients expect that the latest version would be eventually returned if no further `Puts` are performed. Our current prototype implements these consistency models, but ClusterOn can be extended to support other consistency models such as causal consistency [143] as well.

Fault tolerance When the coordinator detects a node failure via periodic heartbeat messages, it launches a new controlet–datalet pair in recovery mode on one of the standby nodes. The new controlet interacts with the coordinator and other existing replica's controlet for data recovery.

5.2.4 APIs and Protocol Parsers

For compatability and modularity, ClusterOn provides a clean set of datalet APIs (between controlet and datalet) and client APIs (between client app and client library), which are consistent with existing I/O interfaces of extant monolithic DOBS. For example, we provide `Enqueue/Dequeue` for message queues, and `Put/Get` for KV stores. Datalet developers can easily adopt them to enable DOBS, and do not need to learn complex new APIs, e.g., as is done in library-based replication solutions such as Vsync [60].

ClusterOn's communication substrate should offer compatibility and be able to understand application protocols to process incoming requests properly. To this end, ClusterOn supports two options. (1) It provides a ClusterOn-defined protocol using Google Protocol Buffers [41]. This option is suitable for new datalets and is preferred due to its ease of use and better programmability. (2) ClusterOn allows developers to provide a parser for their own protocols. This option is mainly available for porting existing datalets such as Redis or SSDB.

5.3 ClusterOn Implementation

We have implemented a prototype of ClusterOn using $\sim 12k$ lines of C++/Python code (counted with CLOC [8]). The rightmost bar of Figure 5.2 summarizes our implementation effort with respect to #LoC. (2) `RC` (*replication and consistency*) implements the main logic of replication and consistency management components. Each new service (e.g., MQ, MS-SC, MS-EC, etc.) requires on average 200–400 LoC to implement the event handler (§5.2.3), and in total constitutes a large portion of the implementation ($\sim 5,600$ LoC, out of which, $1,400$ LoC are for the controlet template). This also implies that it is fairly easy for advanced developers to build new services on top of ClusterOn. (3) `CO` (*coordinator*) consists of two parts: a Python-based failover manager that directly controls the data recovery as well as storing of metadata such as topology information and data mapping table using ZooKeeper [121]. (4) `UT` (*utility*) implements event pipelining, threading, and the lock server client functionality (currently ClusterOn provides two lock server options—ZooKeeper-based [61] and Redlock-based [48] as described in §5.2.3). This part accounts for $\sim 1,600$ LoC. (5) `CL` (*client lib*) integrates our Google Protobuf based protocol into a in-memory KV store client library libmc [34]. We add/modify $\sim 1.5k$ lines of C++ code.

Limitations Our current implementation does not support distributed transactions. While distributed transaction support is still not common in today's NoSQL or KV stores [64], ClusterOn can be extended to support this feature by adding distributed consensus such as Paxos [132] or Raft [151] to the controlet. We are currently working on supporting transactional operations in ClusterOn. Additionally, our ongoing efforts to improve ClusterOn includes adding network kernel stack bypassing [123, 140], memory sharing between controlet and datalet, and caching.

5.4 Evaluation

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

5.4.1 Ease of DOBS Development

First, we evaluate the utility of ClusterOn in simplifying the development of distributed KV stores, based on five different datalets. As described in §5.2.4, in order to glue a datelet into ClusterOn provided control plane, a datalet developer should provide a protocol parser. Figure 5.2 quantifies the engineering effort, in terms of #LoC, needed to develop the protocol parser for each datalet.

New datalets We implement three new datalets that all share a ClusterOn-defined protocol parser (§5.2.4 option 1) built using Google Protocol Buffers [41]. *vHT* (107 LoC) is a simple in-memory hash table, and require much less LoC compared to a similar system, repcached [49] ($\sim 6k$ LoC) that is a replicated Memcached without failover. *vLog* (286 LoC) is a persistent log-structured store that uses *vHT* as the in-memory index. We compare *vLog* with FAWN [66] that has $\sim 7k$ LoC. ClusterOn allows *vLog* to support a rich set of design options, whereas FAWN only supports chain replication based SC. *vMT* (98 LoC) is a Masstree [21][3] based datalet, which supports range query. This is also significantly smaller than $\sim 13k$ LoC for HyperDex [109]. Moreover, in our datalets, $\sim$90% of code is shared and account for common functionalities such as networking/messaging (NW) and threading/event handling (UT).

Existing datalets For existing datalets, we port SSDB [55] and Redis [43] using a simple text-based protocol parser (§5.2.4, option 2) using 333 LoC.

Overall, it took us less than one person day to design and develop (coding & testing) each datalet. This underscores ClusterOn's ability to ease DOBS development.

5.4.2 Experimental Setup

Testbeds and configuration We perform our evaluation on Google Cloud Engine (GCE) and a local testbed. For larger scale experiments, 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.

[3]Note this is an embedded version of Masstree [21] implemented based on the original work [144].

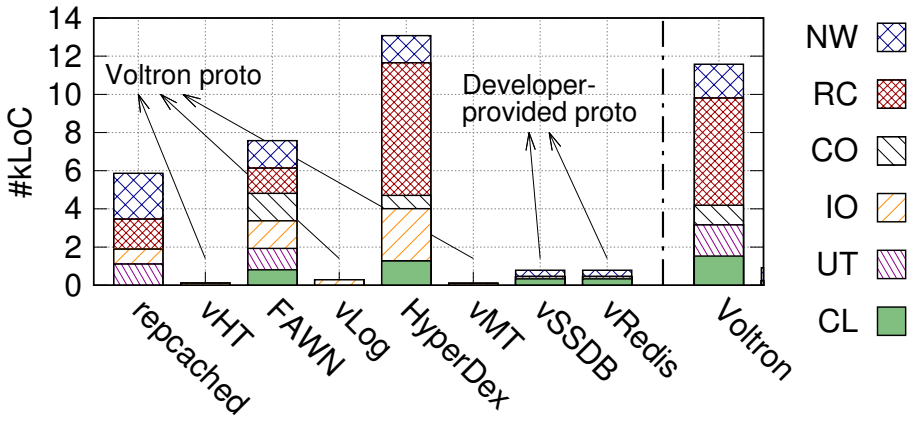
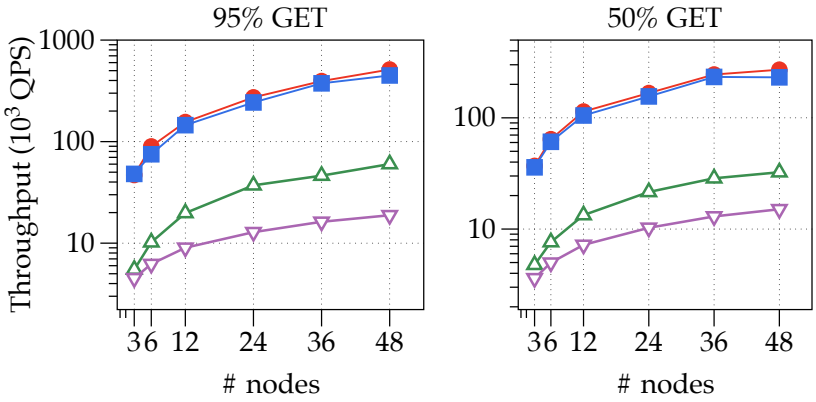


Figure 5.2: Component #LoC of prototyping monolithic distributed KV stores vs. using ClusterOn. All monolithic prototypes, ClusterOn datalets, and the ClusterOn framework are implemented using C++. vHT: ClusterOn-based hash table, vLog: ClusterOn-based log, vMT: ClusterOn-based Masstree [21], vSSDB: ClusterOn-based SSDB [55], vRedis: ClusterOn-based Redis [43].

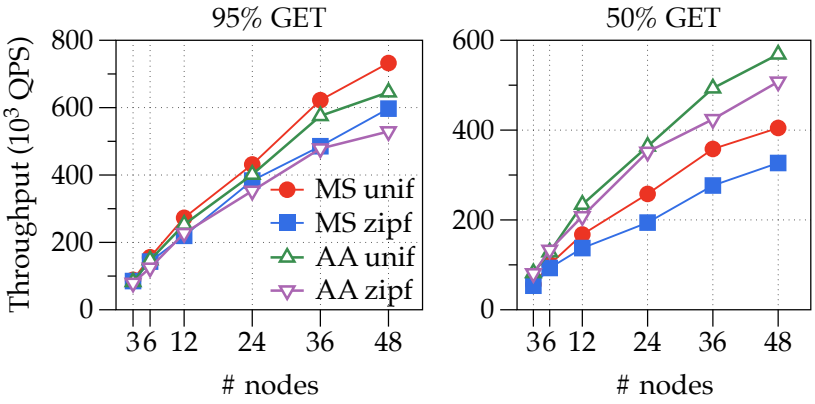
For performance stress test, we use a local testbed consisting of 12 physical machines, each equipped with 8 2.0 GHz Intel Xeon cores, 64 GB memory, with a 10 Gbps network interconnect. The coordinator is a single process (backed-up using ZooKeeper [121] with a standby process as follower) configured 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) [101], 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.



(a) Strong Consistency. On log scale.



(b) Eventual Consistency.

Figure 5.3: ClusterOn scales *tHT* horizontally.

5.4.3 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 [55] and tMT [21], as representatives of existing persistent KV stores.

Figure 5.3 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.

5.5 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 6

Storage for Container Distribution

6.1 Introduction

Container management frameworks such as Docker [12] and CoreOS Container Linux [9] have established containers [147, 158] as a lightweight alternative to virtual machines. These frameworks use Linux *cgroups* and *namespaces* to limit the resource consumption and visibility of a container, respectively, and provide isolation in shared, multi-tenant environments at scale. In contrast to virtual machines, containers share the underlying operating system kernel, which enables fast deployment with low performance overhead [112]. This, in turn, is driving the rapid adoption of the container technology in the enterprise setting [62].

The utility of containers go beyond performance, as they also enable a *microservice* architecture as a new model for developing and distributing software [38, 39, 65]. Here, individual software components focusing on small functionalities are packaged into container *images* that include the software and all dependencies required to run it. These *microservices* can then be deployed and combined to construct larger, more complex architectures using lightweight communication mechanisms such as REST or gRPC [24].

To facilitate the deployment of microservices, Docker provides a *registry service*. The registry acts as a central image repository that allows users to publish their images and make them accessible to others. To run a specific software component, users then only need to "pull" the required image from the registry into local storage. A variety of Docker registry deployments exist such as Docker Hub [13], IBM Bluemix Container Registry [31], or Artifactory [4].

The registry is a data-intensive application. As the number of stored images and concurrent client requests increases, the registry becomes a performance bottleneck in the lifecycle of a container [117, 126, 150]. Our estimates show that the widely-used public container registry, Docker Hub [13], stores at least hundreds of terabytes of data, and grows by about 1,500 new public repositories daily, which excludes numerous private repositories and image

updates. Pulling images from the registry of such scale can account for as much as 76% of the container start time [117]. Several recent studies have proposed novel approaches to improve Docker client and registry communication [117, 126, 150]. However, these studies suffer from a serious shortcoming—they used unreasonably small, oversimplified, or synthetic Docker workloads.

In this work, for the first time in the known literature, we perform a large-scale and comprehensive analysis of real-world Docker registry workload. To achieve this, we started with collecting long-span production-level traces from five datacenters in IBM Bluemix container registry service. IBM Bluemix serves a diverse set of customers, ranging from individuals, to small and medium businesses, to large enterprises and government institutions. Our traces cover all availability zones and many components of the registry service over the course of 75 days, which totals to over 38 million requests and accounts for more than 181.3 TB of data transferred.

We sanitized and anonymized the collected traces and then created a high-speed, distributed, and versatile Docker trace replayer. To the best of our knowledge this is the first trace replayer for Docker. To facilitate future research and engineering efforts, we release both anonymized traces and the replayer for public use. We believe our traces can provide valuable insights into container registry workloads across different users, application, and datacenters.

We also performed comprehensive characterization of the large 75-day traces across several dimensions. We analyzed the request ratios and sizes, the parallelism level, the idle time distribution, and the burstiness of the workload, among other aspects. During the course of our investigation, we made several insightful discoveries about the nature of Docker workloads. We found, for example, that the workload is highly read-intensive comprising of 90-95% pull compared to push operations. Given the fact that our traces come from several datacenters, we were able to find both common and divergent traits of different registries. For example, our analysis reveals that the workload not only depends on the purpose of the registry but also on the life time of the registry service. The older registry services show more predictable trends in terms of access-patterns and image popularity. Our analysis, in part, is tailored to exploring the feasibility of caching and prefetching techniques in Docker. In this respect, we observe that 25% of the total requests are for top 10 repositories and 12% of the requests are for top 10 layers. Moreover, 95% of the time is spent by the registry in fetching the image content from the backend object store. Finally, based on our findings, we derive some design implications for container registry services.

Specifically, in this work we make the following contributions:

- We collect and anonymize large-scale production-level traces for IBM Bluemix container registry that supports a breadth of representative container applications and uses.
- We design and implement a trace replayer for Docker. The traces and the replayer are publicly available at `http://research.cs.vt.edu/dssl/drtp/`.

- Using the traces and the replayer, we perform several comprehensive analysis of Docker registry workload and infer a number of crucial insights about container workloads.
- Based on our analysis, we propose optimizations that reduce request latencies for the Docker registry.

6.2 Background

Docker [12] is a container management framework that facilitates the creation and deployment of containers. Each Docker container is spawned from an *image*—a collection of files sufficient to run containerized application. For example, an image that packages the Apache web server contains all dependencies required to run the server. Docker provides convenient APIs to combine files in an image and run containers from images on end hosts. Each end host runs a daemon process which accepts and processes these API calls from Docker clients.

Images are further divided into *layers*, each consisting of a subset of the files in the image. The layered model allows images to be structured in sub-components which can be shared by other containers on the same host. For example, a layer may contain a certain version of the Java runtime environment and all containers requiring this version can share it from a single layer, reducing storage and network transfer utilization.

In the following sections we describe Docker registry service (§6.2.1) and explain how users interact with the registry by pulling (§6.2.2) and pushing (§6.2.3) images.

6.2.1 Docker Registry

To ease their distribution, images are kept in an online *registry*. Registry acts as a storage and content delivery system, holding named Docker images, available in different tagged versions. Some popular Docker registries are Docker Hub [13], Quay.io [42], Artifactory [4], Google Container Registry [23], and IBM Bluemix Container Registry [31].

Figure 6.1 shows the basic structure of a registry and how users interact with it. Users create *repositories* which hold images for a particular application, e.g. Redis or WordPress. Some repositories contain not a user application but rather a basic operating environment, e.g. Ubuntu or CentOS. Images in such repositories are typically used for building other application images. Images in a repository can have different versions, identified by *tags*. The combination of user name, repository name, and tag defines the name of an image and identifies it uniquely.

Users add new images or update existing ones by pushing to the registry and retrieve images by pulling from the registry using Docker daemon API. The information about which layers constitute a particular image is kept in a metadata file called *manifest*. The manifest also

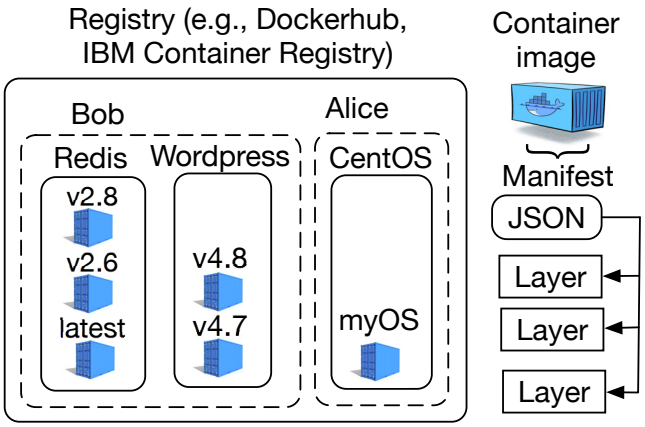


Figure 6.1: On the left: relationship between registry, users (Bob and Alice), repositories (Redis, Wordpress, CentOS), and tagged images (v2.8, latest, v4.8, myOS, etc.); On the right: Docker image structure.

describes other image settings such as target hardware architecture, executable to start in a container, and environment variables. When an image is pulled, only the layers that are not already available locally are transferred over the network.

Content Digest In this study we use Docker's version 2 API which relies on the concept of content addressability. Each layer has a content addressable identifier called *digest*, which uniquely identifies a layer by taking a collision-resistant hash of its bytes (SHA256 by default). Such an identifier can be independently calculated by a Docker client to verify the contents of a layer after pull. Another advantage of content addressability is that it allows to efficiently check whether two layers are identical and deduplicate them for sharing between different images.

6.2.2 Pulling an Image

Clients communicate with registry using RESTful HTTP API. To retrieve an image from the registry, a user sends `pull` command to the local Docker daemon. The daemon then fetches the image manifest by issuing a `GET <name>/manifests/<tag>` request, where `<name>` defines user and repository name while `<tag>` contains image tag.

A manifest typically contains name, tag, fsLayers, and signature fields. Client first verifies manifest signature to ensure names and layers are valid. Then the daemon uses the digests from the fsLayers field to download individual layers that are not already in local storage but still in the registry. Docker client checks layers' availability in registry using `HEAD <name>/blobs/<digest>` requests. Layers are stored in the registry as compressed tar-

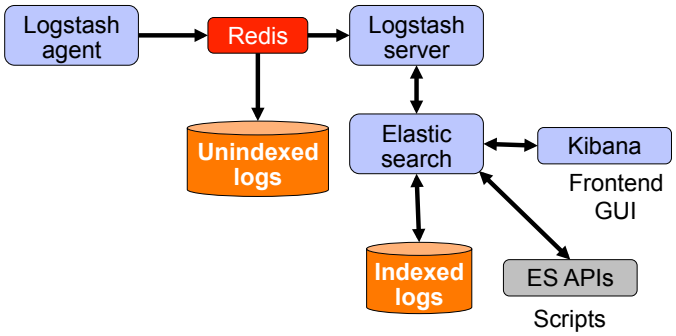


Figure 6.2: ELK stack used for logs collection.

balls ("blobs" in Docker terminology) and are pulled by issuing a `GET <name>/blobs/<digest>` request. Registry can redirect layer requests to a different URL, e.g. , to an object store that stores the layers. In this case, Docker client downloads the layers directly from the new location. By default, the daemon downloads and extracts three layers in parallel.

6.2.3 Pushing an Image

To upload a new image to the registry or update an existing one, clients send a `push` command to the daemon. Pushing works in reverse order compared to pulling. After creating the manifest locally the daemon first pushes all the layers and then the signed manifest.

Docker checks if a layer is already present in the registry by issuing a `HEAD <name>/blobs/<digest>` request. If the layer is absent, its upload starts with a `POST <name>/blobs/uploads/` request to the registry which returns a URL containing `<uuid>`—unique upload identifier—that the client can use to transfer the actual layer data. Docker then uploads layers using *monolithic* or *chunked* transfers. Monolithic transfer uploads the entire data of a layer in a single request via `PUT <name>/blobs/uploads/<uuid>?digest=<digest>`. To carry out chunked transfer, Docker specifies a byte range in the header along with corresponding part of the blob using `PATCH <name>/blobs/uploads/<uuid>` request. Then Docker submits a final `PUT` request with a digest parameter. After all layers are uploaded, client uploads the manifest.

6.3 Tracing Methodology

To collect traces from the Bluemix registry, we gained access to the system's logging service (§6.3.1). The logging service collects request logs from the different system components and the log data contains a variety of information, such as the requested image, the type of request and a timestamp (§6.3.2). This information is sufficient to carry out our analy-

```
{
    "host": "579633fd",
    "http.request.duration": 0.879271282,
    "http.request.method": "GET",
    "http.request.remoteaddr": "40535jf8",
    "http.request.uri": "v2/ca64kj67/as87d65g/blobs/b26s986d",
    "http.request.useragent": "docker/17.04.0-ce go/go1.7.5..)",
    "http.response.status": 200,
    "http.response.written": 1518,
    "id": "9f63984h",
    "timestamp": "2017-07-01T01:39:37.098Z"
}
```


Figure 6.3: Sample of anonymized data.

sis. Besides the traces, we also provide a trace replayer (§6.3.3) to evaluate our proposed optimizations.

6.3.1 Logging Service

As shown in Figure 6.2, logs are centrally managed using an "ELK" stack (ElasticSearch, Logstash and Kibana). A Logstash agent on each server ships logs to one of the centralized log servers, where they are indexed and added to an ElasticSearch cluster. The logs can then be queried using the Kibana web UI, or using the ElasticSearch APIs directly.

The logging service consists of four main components: **Logstash agent** is a Java process, which runs on each server and collects local logs. It then sends the logs to Redis [47] on any of the log servers in round-robin fashion to balance load. **Redis** is an in-memory key/value store. It is used to buffer log messages in memory, to help smooth out any performance spikes. The **Logstash server** reads log messages from Redis, parses them to extract the fields (e.g. the timestamp) and sends them to ElasticSearch to be indexed. **ElasticSearch** stores the logs as JSON documents (one document per log message). Elastic search is a scalable and reliable text-based search system which allows to run full text and structured search queries against the log data. Each AZ has its own ElasticSearch setup deployed on five to eight nodes. Each AZ collects around 2 TB daily of log data that includes system usage, health information, logs from different components etc. Collected data is indexed by time.

6.3.2 Collected Data

For trace collection we use elastic search APIs to pull data from the ElasticSearch setup of each AZ. We pull data for "Registry", "Nginx", and "Broadcaster" components as shown in Figure 6.4 and filter all requests that relate to pushing and pulling of images, i.e. `GET`, `PUT`, `HEAD`, `PATCH` and `POST` requests. Table 6.1 shows the characteristics of collected traces. The total amount of our traces spans seven availability zones and a duration of 75 days from

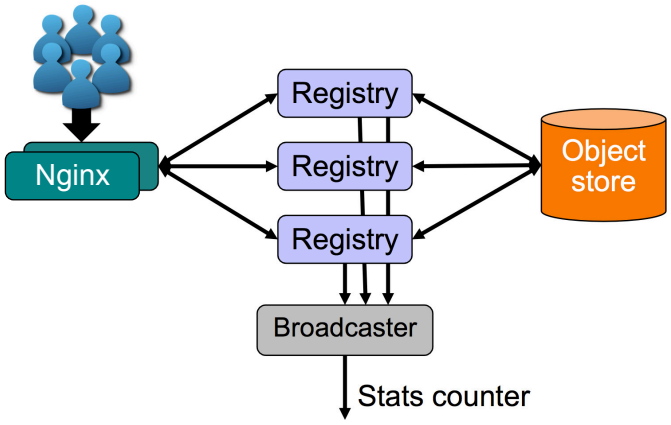


Figure 6.4: IBM Bluemix Registry architecture. Nginx receives users requests and forwards them to registry servers. Registry servers fetch data from the backend object store and reply back.

Aavailablity Zone	Duration (days)	Trace data (GB)	Filtered and anonym. (GB)	Requests (millions)	Data ingress (TB)	Data outgressa (TB)	Images pushed (1,000)	Images pulled (1,000)	Up since (mm/yy)
Dallas (dal)	75	115	12	20.85	5.5	107.5	356	5,000	06/15
London (lon)	75	40	4	7.55	1.7	25.0	331	2,200	10/15
Frankfurt (fra)	75	17	2	1.80	0.4	3.3	90	950	04/16
Sydney (syd)	65	5	0.5	1.03	0.29	1.87	105	360	04/16
Staging (stg)	65	25	3.2	5.90	2.41	29.2	327	1,560	-
Prestaging (prs)	65	4	0.5	0.75	0.23	2.45	65	140	-
Development (dev)	55	2	0.2	0.34	0.01	1.44	15	70	-

Table 6.1: Characteristics of studied data. dal and lon were migrated to v2 in April 2016.

06/20/2017 to 09/02/2017. This results in a total of 208 GB of trace data containing over 38 million requests, with more than 180TB of data transferred in them (data ingress/outgress). Next, we combine the traces from these components by matching the incoming HTTP request identificator across all three components. Then we remove redundant fields to shrink the trace size and in the end we anonymize them. Total size of anonymized traces is 22.2 GB.

Figure 6.3 demonstrates a sample trace record. In the anonymized trace `host` field shows the anonymized registry server which served the request, `http.request.duration` shows the response time of the request, `http.request.method` is HTTP request method (e.g., `PUT`, `GET`), `http.request.remoteaddr` is anonymized remote client IP address, `http.request.uri` is anonymized requested url, `http.request.useragent` field shows the Docker client version used to make this request, `http.response.status` shows the HTTP response code for this request, `http.response.written` shows amount of data that was received or sent, `id` shows the unique identificator of this request, and `timestamp` field shows the request arrival time in UTC timezone.

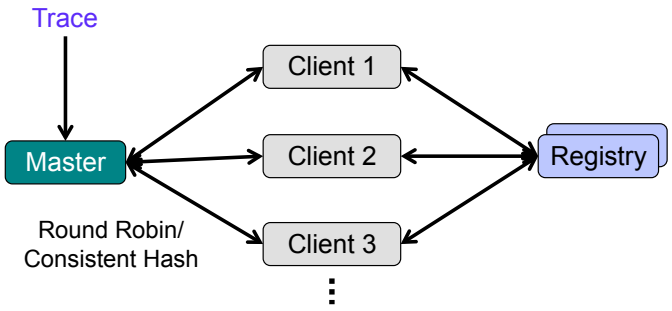


Figure 6.5: Trace replayer. Master parses the trace and forwards request to one of the clients either in round robin or applying consistent hash to the `http.request.remoteaddr` field in the trace.

6.3.3 Trace Replayer

To study collected traces further and use them to evaluate various registry optimizations and improvements we designed and implemented a trace replayer. The trace replayer can operate in two modes to perform two types of analysis: 1) performance analysis of a large scale registry setup and 2) static analysis of traces.

Performance analysis Trace replayer consists of a Master node and multiple client nodes as shown in Figure 6.5. Master node parses the anonymized trace file one request at a time and forwards it to one of the clients. Requests are forwarded to clients in either round robin fashion or by doing a consistent hashing on the `http.request.remoteaddr` field in the trace. By using consistent hashing, trace replayer maintains the request locality to ensure all HTTP requests corresponding to one image push or pull are generated by the same client node as they were seen by the original registry service. In some cases this option may generate workload skewness as some of the clients issue more requests than others. This method is useful for large-scale testing with many clients.

Clients are responsible for issuing the HTTP requests to registry setup. For all `PUT` layer requests, client generates a random file of corresponding size and transfers it to the registry. As the content of newly generated file is not same as the content of the layer seen in the trace, the digest/sha256 is going to be different for the two. Hence, upon successful completion of the request, client replies back to master with latency as well as the digest/sha256 of the newly generated file. Master node keeps track of the mapping between the digest in the trace and its corresponding newly generated digest. For all the future `GET` requests for this layer, master issues request for the new digest instead of the one seen in trace. For all the `GET` requests client replies back with the just the latency to complete the request.

Trace replayer runs in two phases: warmup and actual testing. In warmup phase master iterates over the `GET` requests to make sure that corresponding manifests and layers already

exist in the registry setup. Whereas, in testing phase all requests are issued in the same order as seen in the trace file. The requests are issued by trace replayer as fast as it can. The master side of trace replayer is multithreaded where each client's progress is tracked in a separate process. Once all clients finish their jobs aggregated throughput and latency is calculated. Per request latency and per client latency and throughput is recorded separately.

Static analysis In this mode, clients do not forward the request to the registry instead they reply back to master node after simulating the requested operation. This mode is useful to perform static analysis of the traces. Clients can simulate different caching policies to find out the effect of using different cache sizes. In Section §6.5.3 and Section §6.5.4 we use this mode to perform caching and prefetching analysis.

6.4 Workload Characterization

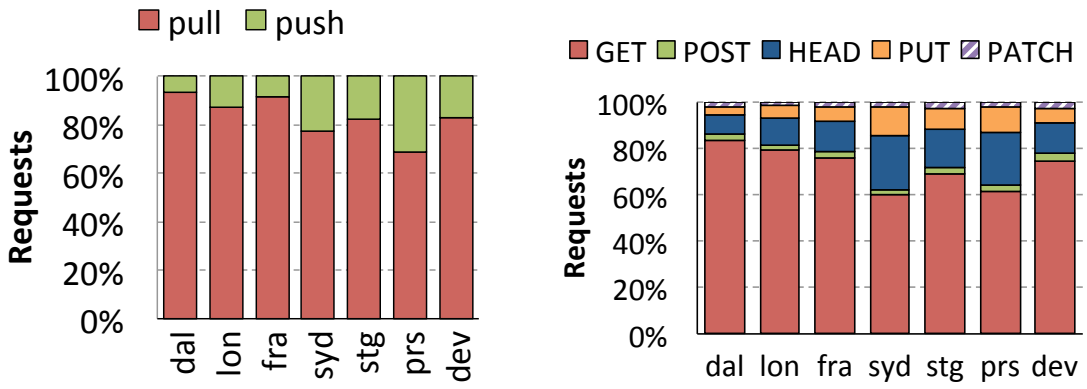
To determine possible registry optimizations, such as caching, prefetching, efficient resource provisioning, and site-specific optimizations, we center our workload analysis around the following five questions:

- What is the general workload the registry serves? What are request type and size distributions? (§6.4.1)
- Do load and response times vary between production, staging, pre-staging, and development deployments? (§6.4.2)
- Is there spatial locality in registry requests? (§6.4.3)
- Do any correlations present among subsequent requests? Can future requests be predicted? (§6.4.4)
- What are the workload's temporal properties? Are there bursts and is there any temporal locality? (§6.4.5)

6.4.1 Request Analysis

We start with the basic request analysis to understand most fundamental properties of the registry's workload.

Request type distribution Figure 6.6(a) shows the ratio of images pulled from vs. pushed to the registry. As expected, registry workload is read-intensive. For `dal`, `lon`, and `fra`, we observe that 90%–95% of requests are pulls (i.e. reads). `Syd` exhibits a lower pull ratio of 78% because it is a newer installation and, therefore, it is being populated more intensively than mature registries. Non-production registries (`stg`, `prs`, `dev`) also demonstrate a lower



(a) Push vs. Pull.

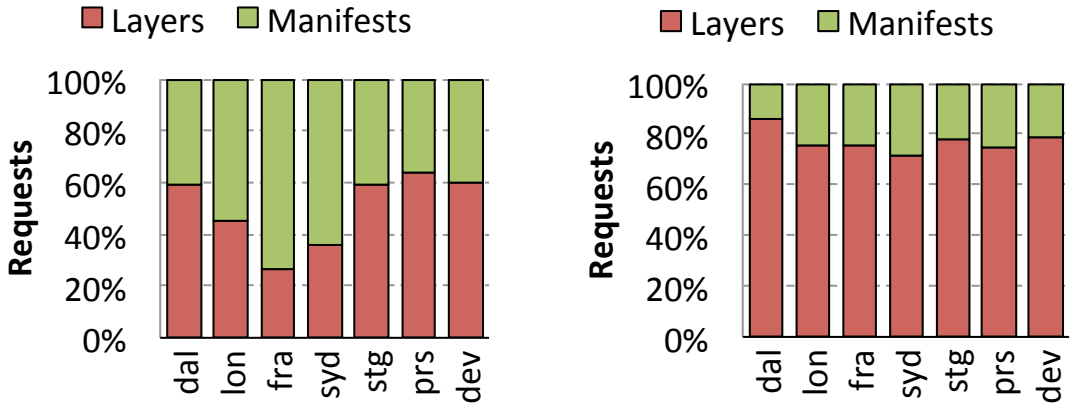
(b) Request Ratio.

Figure 6.6: Image pull vs. push ratio, and distribution of HTTP requests served by registry.

(68–82%) rate of pulls than production registries, due to higher image churn rate. Each push or pull consists of a sequence of HTTP requests as discussed in § 6.2. Figure 6.6(b) shows the distribution of different HTTP requests served by the registry. All registries receive more than 60% of `GET` requests and a 10%–22% of `HEAD` requests. `PUT` requests are 1.9–5.8× more common than `PATCH` requests because `PUT`s are used for uploading manifests (in addition to layers) and many layers are small enough to be uploaded in a single request.

Figure 6.7(a) and Figure 6.7(b) show the manifest vs. layer ratio for pull and push image requests, respectively. For pulls we observe that, except for `syd` and `fra`, there is the same amount or more requests retrieving layers than manifests, which is logical as a single manifest refers multiple layers. Our investigation showed that the divergent behavior of `syd` and `fra` is caused by their clients trying to pull images that they have already pulled in the past. This results into many `GET` requests to the manifests without subsequent `GET` requests to the layers. For pushes, we see that accesses to layers dominate accesses to manifests.

Request size distribution Figure 6.8 shows the CDF of manifest and layer sizes for `GET` and `PUT` requests. In Figure 6.8(a) we observe that about 65% of the layers are smaller than 1 MB and around 80% are smaller than 10 MB. In Figure 6.8(b), we find that the typical manifest size is around 1 KB for all AZs except for `lon` where 50% of the GET requests are for manifests larger than 10 KB. For `lon`, lots of requests are for manifests that are compatible with the older Docker version, hence making their size larger. We observe similar trends for `PUT`s for all the AZs (not shown in the Figures).



(a) Pull. (b) Push.

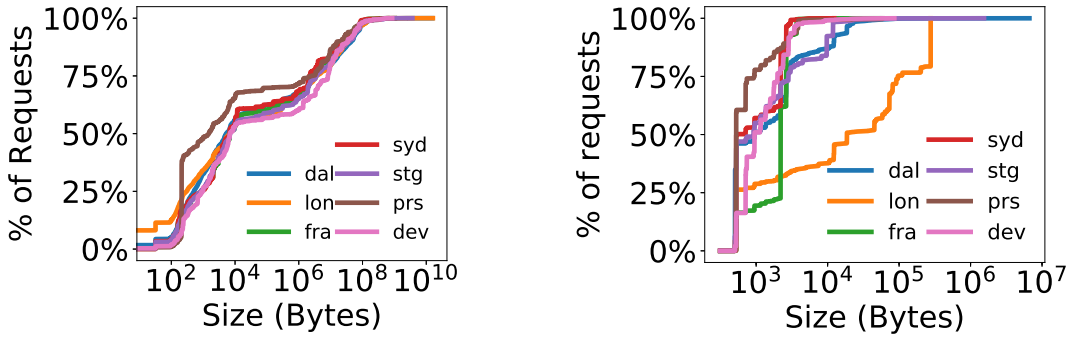
Figure 6.7: The ratio of requests that access either an image manifest or a layer. We include `GET` requests in pull count, while pushes include `PUT` *or* `HEAD` requests, to account for attempts to upload the layers that already present in the registry.

6.4.2 Registry Load and Response Time

Load distribution Figure 6.9 shows the CDF of received requests per minute. `Dal` has the highest overall load and serviced at least 100 requests per minute more than 80% of the time. `Lon` and `stg` are second and third, followed by `fra`, `syd`, `prs`, and `dev`, in descending order. This trend is consistent across the different request types (not shown in the Figure). The ordering of AZs by the load yields two main observations. First, development and pre-staging registries experience low utilization. `Dev`, for example, does not receive any requests 57% of the time. Second, registry load increases with its lifetime. In our traces `dal` and `lon` have been running the longest while `fra` and `syd` have only been started recently.

Response time distribution Figure 6.10 shows the CDFs of response time of different requests to layers and to manifests. As `dal` is the highest loaded AZ, its request response times are higher compared to other AZs. More than 60% of the `GET` layer requests take more than one second to finish. For the top 25% of requests we see a response time of ten seconds and higher. `Fra`, `syd`, `prs`, and `dev` are not highly loaded, so they have the lowest latency in serving the `GET` layer requests. `POST`, `PATCH`, and `PUT` layer requests follow similar trends. However, `PATCH` requests are visibly slower than `GET`s and `PUT`s as they carry more data. We also analyzed the dependency of response time from the layer size (not shown in the Figures) and concluded that response time remains the same for layers smaller than 1 MB, and then starts to grow linearly.

Figure 6.10(d) and 6.10(e) show the response time distributions for `PUT` and `GET` requests to manifests, respectively. Since manifests are smaller and cached, we observe significantly



(a) Layers. (b) Manifest.

Figure 6.8: CDF of manifest and layer sizes for GET requests.

smaller and more stable latencies than that of requests serving layers. One interesting observation is that `lon` has the highest response time when serving manifests (300-400 ms more than `dal`). This is because `lon` serves manifests with larger sizes compared to other AZs. This is also consistent with the results shown in Figure 6.8(b). For the `PUT` manifest requests we observe more uniform trend across the AZs as size of the new manifests is same for all the AZs.

6.4.3 Popularity Analysis

In this section we study the popularity of layers, manifests, users, repositories, and clients to answer whether image accesses are skewed and produce hot-spots.

Popularity distribution Figure 6.11 shows the CDF of the access rate of layers, manifests, repositories, and clients. Figures 6.11(a) demonstrates that there is a heavy skew in layer accesses. For example, the 1% most frequently accessed layers in `dal` account for 42% of all requests while in `syd` this increases to 59%. However, requests to the `dev` and `prs` sites are almost evenly distributed. The reason is that testing developers frequently need to push or pull images not available neither at registry nor at client side. We also observe that the younger AZs experience a higher skew compared to the older AZs. We believe this is due to the fact that accesses become more evenly distributed over a long period of time.

For manifest accesses (Figure 6.11(b)) skew is more significant than for layers. This confirms that there are indeed hot *images* which can benefit from caching. Repository accesses (Figure 6.11(c)) reflect this fact but show slightly less skew as manifests are contained in repositories (and hence accesses to different manifests may still go to the same repository). The same trend holds for users under which repositories are stored (not shown in Figure 6.11).

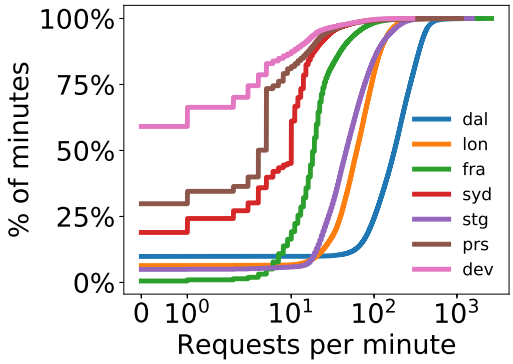


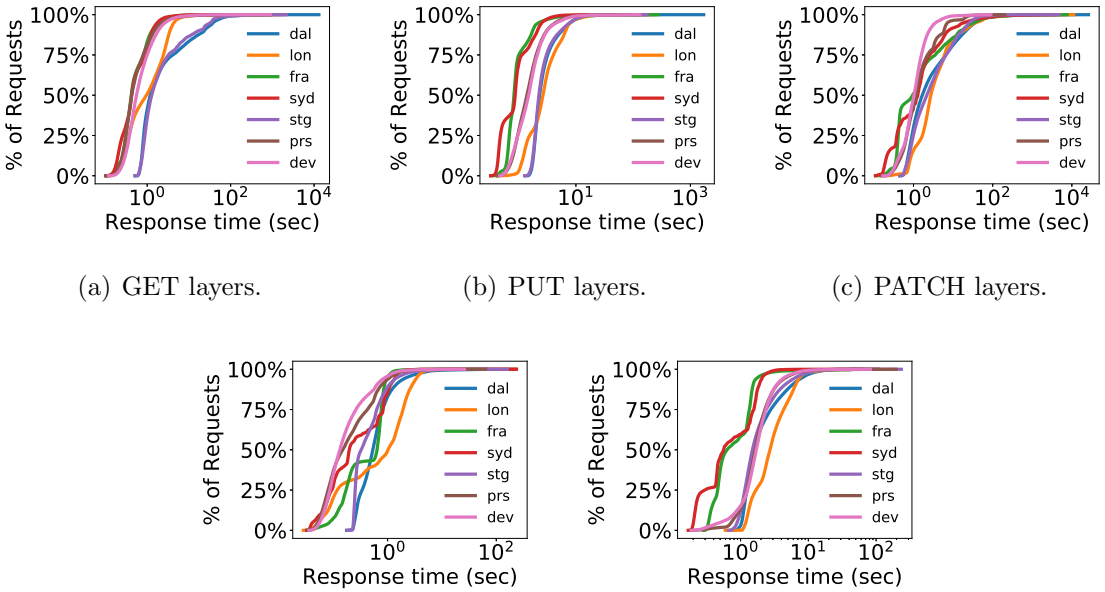
Figure 6.9: CDF of requests per minute.

Furthermore, we find that client accesses are also heavily skewed (Figure 6.11(d)). This means that there are few highly active clients while most of them only submit few requests. This trend is consistent across all AZs. While this does not directly affect the workload, clients can be biased towards a certain subset of images which will contribute to the access skew.

Top-10 analysis To further understand the popularity distribution of registry items, we analyze the top 10 hottest items in each category. Figure 6.12(a) shows the access rates for the top 10 layers, which account for 8%–30% of all accesses depending on the registry. The most popular layer (rank 1) in all AZs absorbs 1–10% of all the requests while in `syd` absorbs 19%. The popularity rate drops rapidly as we move from most popular to tenth most popular layer. The relative amount of accesses for the top 10 layers is the lowest for `dal` is it stores the most layers and experiences the highest amount of requests.

For the top 10 manifests (Figure 6.12(b)), we observe that some container images are highly popular and account for as many as 40% of the requests in `fra` and `syd`, and 60% in `prs`. Note that a manifest is fetched even if the image is already cached at the client side. Hence, a manifest fetch does not necessarily mean that the corresponding layers are fetched (§6.4.4). Similar to Figure 6.11, the skew decreases for repository popularity (Figure 6.12(c)) and user popularity. Part of the reason for the small number of highly accessed images is that registry services in production are tested periodically to monitor their health and performance. For the AZs with a smaller workload (`fra` and `syd`), those test images make up three out of top five most accessed images. We intentionally did not exclude these images from our analysis as they are part of the registry workload.

Figure 6.12(d) shows that the most popular client submit around 15% of the total requests. This excludes `prs` and `dev`, which are used by the registry development team for internal development and testing. These two AZs only have a small number of clients and 2 clients



(a) GET layers. (b) PUT layers. (c) PATCH layers.

(d) GET manifests. (e) PUT manifests.

Figure 6.10: CDF of response time for `GET`, `PUT`, `PATCH` requests to layers and `GET` and `PUT` requests to manifests.

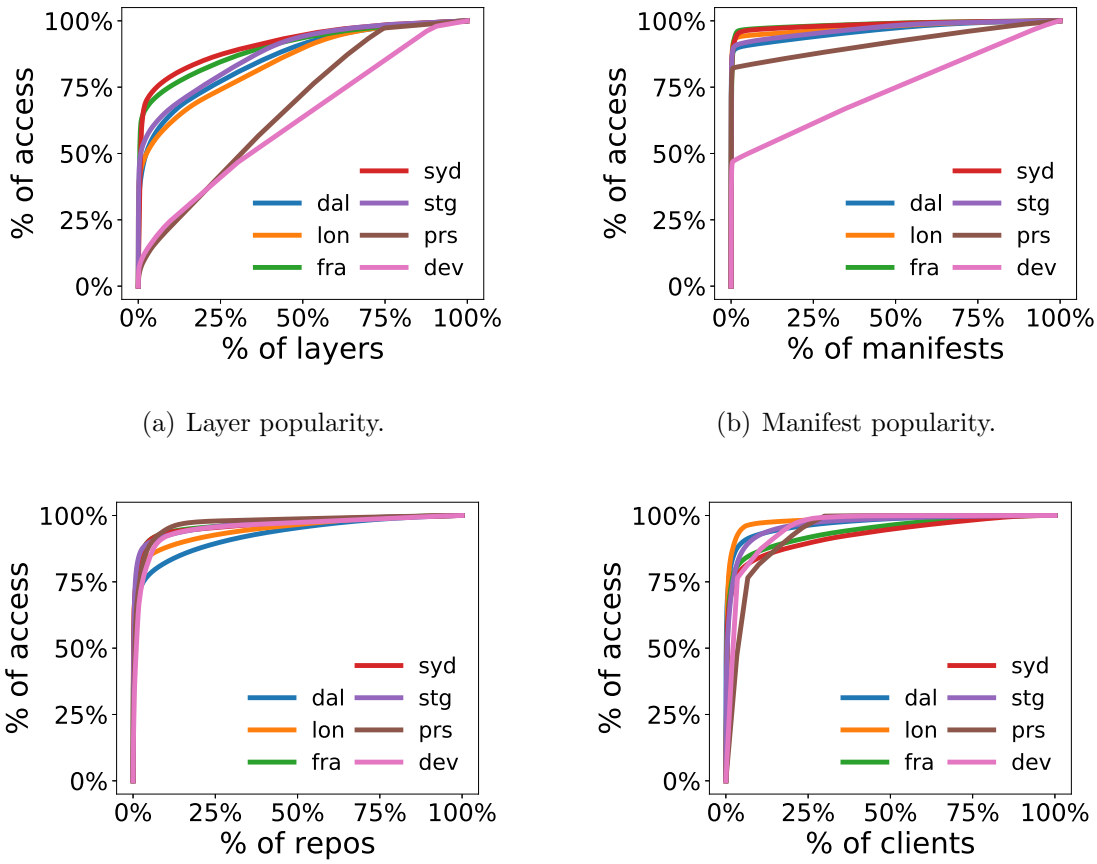
already contribute around 80% of all requests.

Overall, the detailed top-10 analysis shows that while there are few highly popular test images, the popularity of the remaining hot items is decreasing fast and hence, overly small caches will be insufficient to effectively cache data. E.g., based on this results we estimate that 2% cache (compared to the dataset size) can provide 40% and higher hit ratio.

6.4.4 Request Correlation

In this section we investigate whether a `GET` request for a certain manifest always results in subsequent `GET` requests to the corresponding layers. Therefore, we define a client *session* as the duration from the time a client connects until a certain threshold. We varied the threshold from 1 to 10 minutes but could not observe significant differences. However, values less than 1 minute dramatically affect the results as that is less than the typical time, a client takes to pull an image. We set the session threshold to 1 minute and then counted all `GET` layer requests that follow manifest's `GET` within a session.

Figure 6.13(a) shows the CDF of the number of times clients issue the corresponding `GET` layer requests after retrieving a manifest. In most cases, ranging from 96% for `dev` to 73% for



(a) Layer popularity.

(b) Manifest popularity.

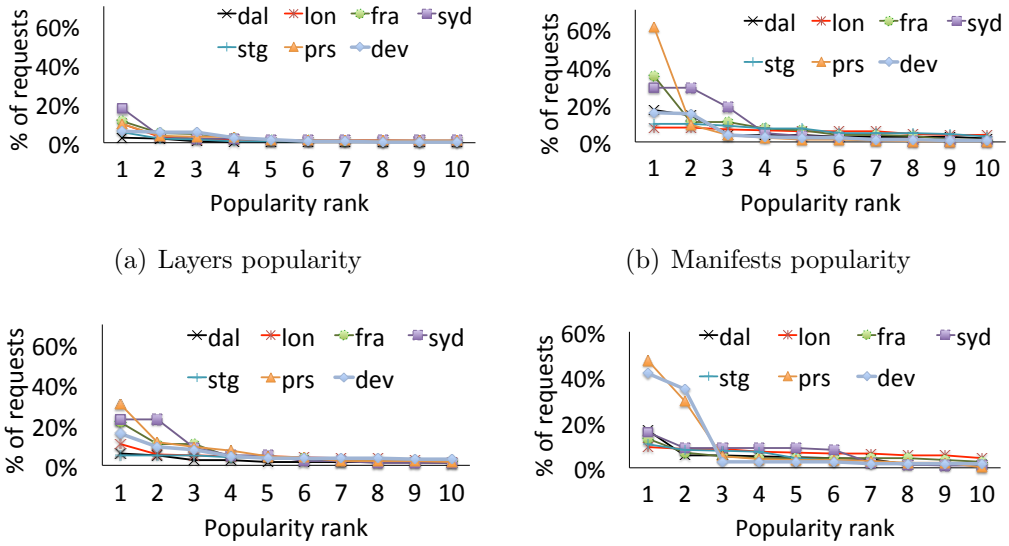
(c) Repository popularity.

(d) Client popularity.

Figure 6.11: CDF of access for layers, manifests, repositories, and clients.

`fra`, `GET` manifest requests are not followed by any subsequent request. The reason is that whenever a client has already fetched an image and then pulls an image, only the manifest file is requested to check if there has been any change in the image. This shows that there is no strong correlation between `GET` manifest and layer requests.

We then focused only on `GET` manifest request that were received within the session of a `PUT` request to the same repository, from which the manifest is fetched (Figure 6.13(b)). This leads to a significant increase in subsequent `GET` layer requests within a session for all production and staging AZs. The manifest requests not followed by `GET` layer requests are due to the fact that clients sometimes pull the same image more than once. Overall, our analysis reveals a strong correlation between `GET` manifest and subsequent layer requests if preceding `PUT` requests are considered.



(a) Layers popularity

(b) Manifests popularity

(c) Repos popularity

(d) Clients popularity

Figure 6.12: Popularity of top ten layers, manifests, repositories, and clients.

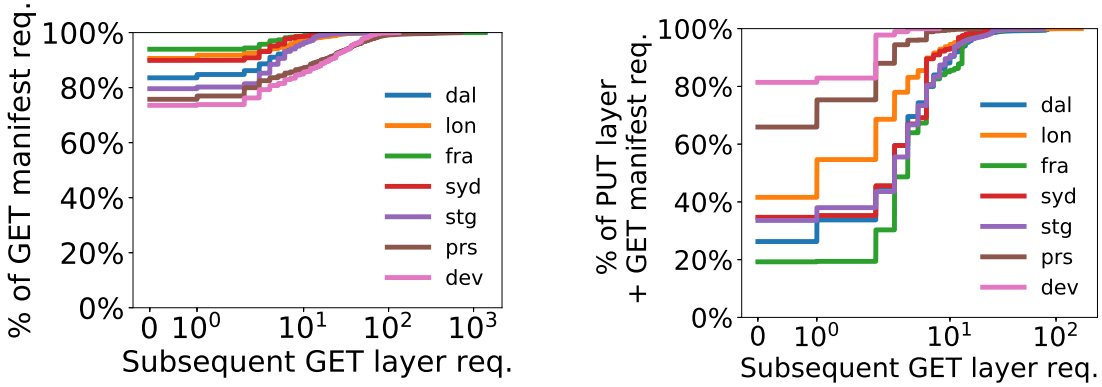
6.4.5 Temporal Properties

In this section we investigate whether the workload shows any temporal patterns.

Client and request concurrency We start with measuring how many clients and requests are *active* at a given point of time. Active client is the client that maintains a connection to the registry, while active request is the request that was received but not yet processed by the registry. Figures 6.14(a) and 6.14(b) show the results for clients and requests, respectively. Overall, the median amount of concurrent clients is low, ranging from 0.6 clients for `dev` to 7 clients for `dal`. However, there are peak periods during which several hundred clients are connected at the same time. We observe a similar trend when looking at parallel requests.

To understand whether these peak periods have diurnal patterns, we plot the average number of requests per hour and day across all traced hours and days in Figures 6.15(a) and Figure 6.15(b). For `dal`, we observe that request numbers are decreasing during the night and over the weekend. While other AZs show a similar trend, it is less pronounced at those sites. This suggests that registry resources can be provisioned statically for hours and days. We plan to explore short-term bursts in the future.

Inter-arrival and idle times Next, we look at request inter-arrival and idle times to study whether the registry experiences longer periods of idleness, during which less resources are required. Inter-arrival time is defined as a difference between the times when the registry receives two subsequent requests. We measure the idle times by monitoring active requests.



(a) Subsequent GET layers per GET manifest.

(b) Subsequent GET layers per PUT layer + GET manifest.

Figure 6.13: Relationship between GET manifest and subsequent GET layer requests.

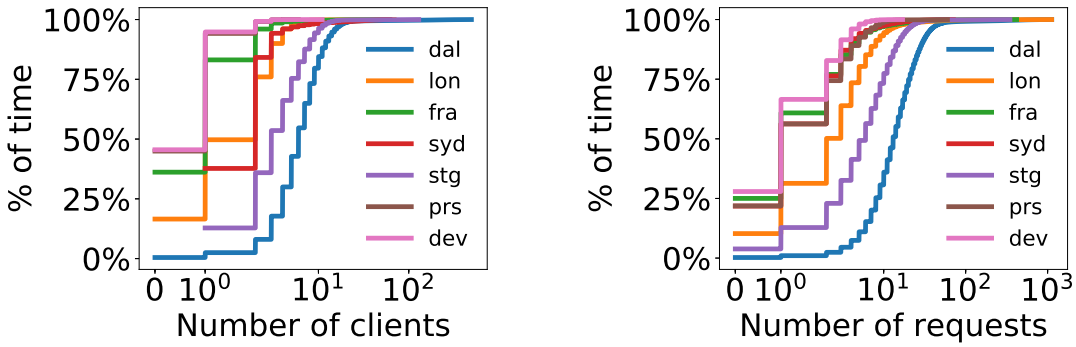
When there are no active requests, we record the time until the next request arrives. This constitutes the idle time.

Figure 6.16(a) shows the inter-arrival times. `dal`, `lon` and `stg` experience the highest request frequency with a 99$^{\text{th}}$ percentile of 3 seconds while for the rest we see 110 seconds. When looking at idle times (Figure 6.16(b)), we observe that idle periods are short and in most cases below 1 second. However, the amount of experienced idle periods varies significantly across AZs. Throughout the entire collection time, `dal` only saw approximately 0.1 million idle periods while `lon` experienced more than 1.5 million. While some AZs experience a large amount of idle periods, their duration is short and hence, they are hard to account for in resource provisioning approaches.

6.4.6 Analysis Summary

We summarize our analysis in seven observations:

- `GET` requests are dominant in all registries and more than half of the requests are for layers, opening an opportunity for an effective layer cache at the registry.
- 65% and 80% of all layers are smaller than 1 MB and 10 MB, respectively, making individual layers suitable for caching.
- The registry load is affected by the registry intended use case and the lifetime of the registry. Younger, non-production registries experience lower loads compared to longer running, production systems. This should be considered when provisioning resources for an AZ to save cost and use existing resources more efficiently.



(a) Client concurrency.

(b) Request concurrency.

Figure 6.14: CDF of client and request concurrency.

- Response times correlate with registry load and hence also depend on the age and the use case of the registry.

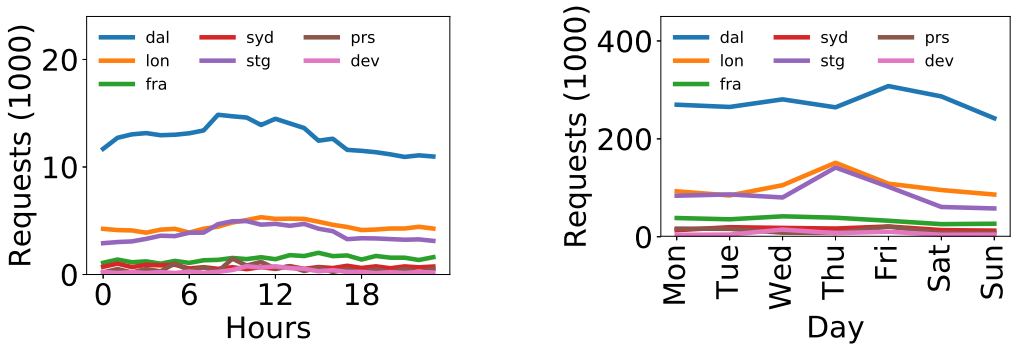
- Registry accesses to layers, manifests, repositories, and by users are heavily skewed. Few extremely hot images are accessed frequently but the popularity drops rapidly. Therefore, caching techniques are feasible but cache sizes should be selected carefully.

- There is a strong correlation between PUT requests and subsequent GET manifest and GET layer requests. The registry can leverage this pattern to prefetch the layers from the backend object store to the cache, significantly reducing pull latencies for the client.

- While there are weak drops in request rates during weekends, we did not find pronounced repeated spikes that can be used to improve resource provisioning.

6.5 Registry Design Improvements

In this section, we use the observations from §6.4 to design two improvements to the container registry: (i) a multi-layer cache for popular layers; and (ii) a tracker for newly pushed layers, which enables prefetching of the newest layers from the backend object store. We evaluate our design using our trace replayer.



(a) Hours of day

(b) Days of week

Figure 6.15: Average number of requests over the tracing period for each hour of the day and day of the week.

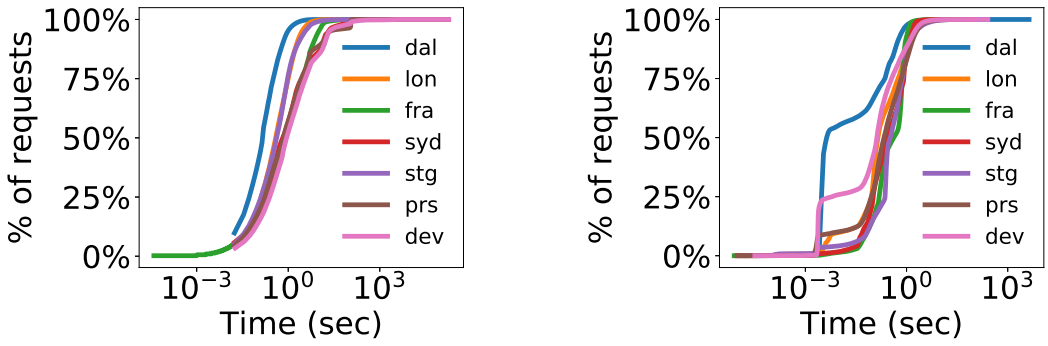
6.5.1 Implementation

We implemented a performance analysis mode for the trace replayer using around 500 LoC of Python. This mode allows us to study the effect of different storage technologies on response latency. We used Bottle [5] for routing requests between the master and clients. Moreover, we used the dxf library [18] for storing and retrieving data to/from Docker v2 registry. For caching and prefetching, we implemented two separate Python modules using around 350 and 300 LoC, respectively. To implement the in-memory storage for layers inside the Docker registry, we modified the Swift storage driver of the registry (about 200 LoC modified/added). Our choice of Swift storage driver is based on the need to store only the small sized layers in memory, and to use Swift storage for larger layers.

6.5.2 Performance Analysis

We analyze the response latency of Docker v2 registry using the following setup. The registry is launched on a 32 core machine with 64 GB of RAM and 512 GB of SSD storage, and the Swift object store runs on a separate set of nodes of similar configuration. The trace replayer is started on additional six nodes (one master and five clients). We made sure that the trace replayer or the object store are never the bottleneck during this analysis. All nodes are connected via 10 Gbps network links. To drive the analysis, the trace replayer is used to replay 10,000 requests from the `dal` trace (August 1^{st}, 2017 starting at 12 am).

Figure 6.17 shows the latency vs. layer size for four different backends: 1) Swift; 2) memory for layers smaller than 1 MB and Swift for rest of the layers; 3) local file system with SSD; and 4) redirection—where the registry replies back with the link to the layer in Swift and



(a) Request inter-arrival time.

(b) Request idle time.

Figure 6.16: CDF of request inter-arrival and idle times

the client then fetches the layer directly from Swift.

We observe that, for small sized layers (i.e. layers less than 1 MB), the response time is the lowest, 0.008 s on average, for the layers stored in memory. This is followed by the SSD, which yields a response time of around 0.013 s for small objects. Swift object store is next with an average response time of 0.07 s. The redirection approach performs the worst for small objects with average response time of 0.11 s. For large size objects, we observe that SSD beats Swift and Redirection and has an average response time of 0.4 s. Moreover, for layers slightly larger than 1 MB, Swift outperforms redirection. However, for very large layers, Swift and redirection approach perform similarly, with average response latencies of 0.63 s and 0.59 s, respectively.

These results highlights the advantage of having a fast backend storage system for the registry, and bring to light the opportunity for caching to significantly improve the performance of the registry.

6.5.3 Two-levels cache

In designing our cache, we chose to exploit the high capacity RAM as well as SSDs that are present in modern server machines. We also observed that a small fraction of layers are huge, and thus caching them in memory is not he best use of the available RAM. Consequently, we design a multi-level cache comprising RAM (for smaller layers) and SSDs (for larger layers). We do not have to worry about cache invalidation. This is because the layers are content addressable and any change in the layer also changes its digest, effectively making it a "new" layer for caching. The older version of the layer is no longer accessed and eventually gets evicted as its popularity plunges.

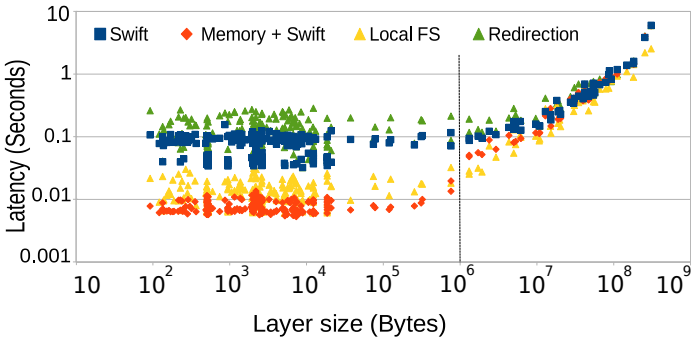


Figure 6.17: Effect of various backend storage technologies on registry performance.

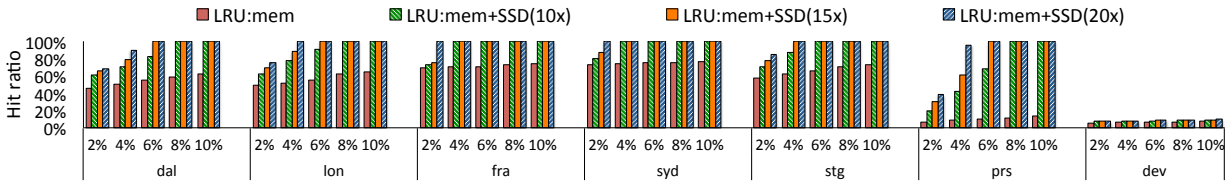


Figure 6.18: Hit ratio of LRU caching policy for both the memory and the SSD level cache.

Hit ratio analysis

In this section, we present a simulation based evaluation of our two-level cache for the registry servers. For these experiments, we mimic the IBM registry server setup. We simulate the same number of servers as there are in each AZ and for each server, we add a memory and SSD cache. The registry servers do not share the cache as Docker v2 registry implementation is non-distributed. However, the setup can be scaled by adding more registry servers behind the Nginx load balancer.

We use the LRU caching policy for both the memory and the SSD level cache. We select cache sizes of 2%, 4%, 6%, 8%, and 10% of the data ingress for each AZ. The data ingress of an AZ is the amount of *new* data stored in that AZ during the 75 days period in which we collected the traces. For the SSD level cache sizes, we select 10×, 15×, and 20× the size of the memory cache. Any object evicted from the memory cache first goes to the SSD cache before it is completely evicted. We store layers smaller than 100 MB in the memory level cache, while bigger layers are stored in the SSD level cache. For our analysis, we iterate over the traces and wait for the cache to be filled, and start calculating the hit and miss ratios upon observing the first eviction from the cache. Given that our traces are long, the first eviction happens quite early relative to the time it takes to replay all traces.

Figure 6.18 shows the results of adding the two-level cache to the registry. We see that for

Algorithm 3: Layers Prefetching Algorithm.

Input: LM_{thresh}: Threshold for duration between PUT layer and subsequent GET manifest requests,
ML_{thresh}: Threshold for duration to keep prefetched layer.

```
while true do
    r ← request received
    if r = PUTlayer then
        /* Create new entry for layer                                          */
        RepoMap[r.repo] ← NewEntry(r.client, r.layer)
        RepoMap[r.repo] ← set LM_timer
        /* When LM_timer > LM_thresh, entry is evicted                         */
    else if r = GETmanifest then
        if r.client not in RepoMap[r.repo] for r.layer then
            RepoMap[r.repo] ← add r.client
            PrefetchedLayers ← prefetch r.layer
            PrefetchedLayers[r.layer] ← set ML_timer
            /* When ML_timer > ML_thresh, layer is evicted                     */
            prefetch++
    else if r = GETlayer then
        if r.layer in PrefetchedLayers then
            serve from PrefetchedLayers[r.layer]
            prefetch_hit++
        else
            serve from object store
```

all the user-driven AZs (e.g. four production AZs and one internally used AZ), adding even a single level of LRU-based memory cache yields a hit ratio of 40% for `dal` with cache size of 2% of ingress data and as high as 78% for `fra` and `syd` with cache size of 10% of ingress data.

Increasing the cache size increases the hit ratio, until it reaches the max of 78%. This is because we only put layers less than 100 MB in the memory cache. However, when we enable the second level cache, we achieve a combined hit ratio of 100% with 6% cache size for `dal` and 4% cache size for rest of the client-driven AZa. We also observe very different results for the non-user-driven AZs `prs` and `dev`. As these two traces represent testing interactions by the registry development team, we do not see any advantage of using the cache in this case.

6.5.4 Prefetching layers

Our second design improvement is to enable prefetching of layers from the backend object store by predicting what layers are most likely to be requested. Therefor, we use our observations of the push-pull relationship established in §6.4.4 to predict what layers to prefetch as shown in Algorithm 3.

We have observed that the incoming PUT requests determine which layers will be prefetched when the registry receives a subsequent GET manifest request. When a PUT is received,

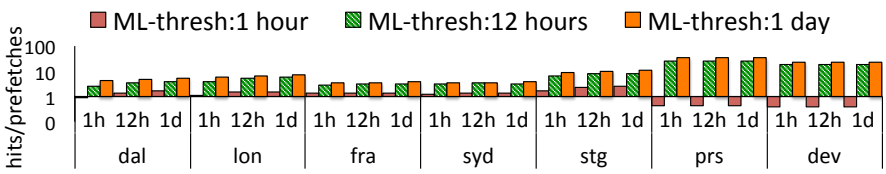


Figure 6.19: Hits/prefetch ratio.

the repository and the layer specified in the request will be added to a *look up* table that includes the request arrival time and the client address. When a GET manifest request is received from a client within a certain threshold LM_{thresh}, the host checks if the *look up* table contains the repository specified in the request. If it is a hit and the client's address is not present in the table, then the address of the client is added to the table and the layer is prefetched from the backend object store. Note that both the amount of time that the entries remain in the *look up* table and how long the layers are kept in the registry side, defined by ML_{thresh}, are configurable.

Hit/Prefetch Analysis

We tested our algorithm using different values for retaining *look up* table entries, LM_{thresh}, and retaining prefetched layers, ML_{thresh}. We use values of 1 hour, 12 hours, and 1 day for each of the threshold parameters. Figure 6.19 shows the results. Single bars represent ML_{thresh} values while groups of bars are assigned to LM_{thresh} values.

On one hand, we find that increasing ML_{thresh} can significantly increase the hit/prefetch ratio. Here, our algorithm is able to better predict the probability that a client will ask for the prefetched layer as layers are retained for a longer period. On the other hand, increasing the retention threshold for the *look up* table entries only marginally increases the hit ratio. This is because the longer an entry persists in the table, the fewer prefetches it serves as the record of clients added to the table increases. We also find that the maximum amount of memory used by `dal`, `lon`, `fra`, `syd`, `prs`, and `dev` is 10 GB, 1.7 GB, 0.5 GB, 1 GB, 2 MB, and 69 MB. We note that for both `prs` and `dev` the maximum amount of memory is low because they experience less activity and therefore contain less PUT requests compared to other cases.

Our analysis shows that it is possible to improve registry performance by adding an appropriate sized cache. For small sized layers, a cache can improve latency time by an order of magnitude. With appropriate sized cache we can achieve above 90% hit ratio. We also show that it is possible to predict the GET layer requests under certain scenario to facilitate prefetching.

6.6 Summary

A Docker registry platform plays a critical role in providing containerizing virtualization services. However, heretofore, the workload characteristics of the Docker registry have remained unknown. This work has presented a dizzying number of views into a very large scale of Docker registries (we have analyzed a set of trace data including 38 million requests spanning over more than 181.3 TB traces during 75 days). Together, these views tell a coherent story of five geographically distributed datacenters housing production Docker registries. This study answers pertinent questions and provides a handful of insights to improve the performance and usage of Docker registries. Moreover, we have implemented a trace replayer which enables detailed and multi-dimensional offline trace analysis. Based on our findings, we proposed several design optimizations by using caching and prefetching, and showed their significant impact on the performance of the registry service. Finally, we have open-sourced our traces and the trace replayer software, which can be used to serve as a solid basis for new research and studies on registry service and container virtualization.

Chapter 7

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 modularity 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, 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

Similarly, MOS posit that compared to using a rigid object store *it is more beneficial to support multi-tenant workloads separately using dynamically configurable finer-grained object stores* on sub-clusters of available resources.

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. This dissertation selects three different application scenarios—cloud object stores, distributed key-value storage systems, and storage for containers—targeting two general modern workloads—internet-scale web workloads and IBM Docker containers workloads.

By performing extensive and deep analysis to understand the issues [67, 72, 77, 78, 128, 129], designing rigorous models [72, 77, 129] and practical tools [77] to characterize complex workload behaviors, and building efficient systems [68, 70, 73, 74, 76, 77, 157] to manage resources and functionalities, this dissertation demonstrates improved efficiency and flexibility at the system level with a broad focus on practical and user-centric metrics.

7.1 Summary

A typical deployment of cloud object stores either opts to use a monolithic configuration or segmented storage setup [57] with a static configuration to handle different types of applications with evolving requirements. From the cloud provider's perspective, supporting dramatically different workloads from different applications (tenants) using a single homogeneous configuration means that optimization opportunities are lost. The situation is further complicated by the fact that due to regular system upgrades and introduction of new storage architectures, data centers hosting the object stores are becoming increasingly heterogeneous [107, 145]. 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 cloud computing model to understand and improve the design of underlying architecture provided by open source cloud solutions, such as OpenStack, to efficiently carry out monitoring and metering tasks. In this work, we focus on how the monitoring data is collected by Cloud providers and present solutions for the open problems. Specifically, we present a deep analysis of how open source cloud solutions, such as OpenStack, facilitates monitoring and metering. Then, by specifically focusing on how the monitoring data is collected by OpenStack and by analyzing the collected data we identify open problems in OpenStack. We also present solutions for those open problems and discuss the challenges therein.

Second, we tackle resource partitioning for object stores. typical deployment of cloud object stores either opts to use a monolithic configuration or segmented storage setup [57] with a static configuration to handle different types of applications with evolving requirements. Using a monolithic configuration setup results in all applications experiencing the same service level, e.g., similar average latency per request, data transfer throughput, and queries per second (QPS). However, different applications entail extremely different latency and throughput requirements. For example, a social networking or photo sharing application requires low latency to support a highly-responsive user experience, whereas backup services can tolerate higher latency but require sustained high throughput. In this work, we have presented an experimental analysis of cloud object store, and proposed a set of rules-of-thumb based on the study. The rules provide practical guidelines for service administrators and online resource managers to better tune object store performance to application needs. The resulting system, MOS, outperforms extant object stores in multi-tenant environments. Furthermore, we build MOS$_{++}$ to enhance MOS by leveraging containers for fine-grained resource management and higher resource efficiency.

This dissertation then 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 dis-

tributed 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 [68], 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.

Finally, this dissertation tackles object storage for containers. This work has presented a dizzying number of views into a very large scale of Docker registries (we have analyzed a set of trace data including 38 million requests spanning over more than 181.3 TB traces during 75 days). Together, these views tell a coherent story of five geographically distributed datacenters housing production Docker registries. This study answers pertinent questions and provides a handful of insights to improve the performance and usage of Docker registries. Moreover, we have implemented a trace replayer which enables detailed and multi-dimensional offline trace analysis. Based on our findings, we proposed several design optimizations by using caching and prefetching, and showed their significant impact on the performance of the registry service. Finally, we have open-sourced our traces and the trace replayer software, which can be used to serve as a solid basis for new research and studies on registry service and container virtualization.

7.2 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.

7.2.1 Scalable Docker Registry

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. For example, to increase the scalability of the Docker registry, the production registry layers need to coalesced together into a single, distributed registry. Similarly, proxy assisted caching requires cooperation between layers. However, cooperative caching opens up several research and design questions to accomodate efficient load balancing.

7.2.2 Edge Computing Framework

IoT holds tremendous advantages for human society. As computing capacity of Edge devices is increasing, trend to perform computing on the Edge devices is also gaining traction. However, existing Edge computing frameworks lacks in term of supporting multi-tennant applications on heterogeneous resources in truly distributed manner. There is extreme need of a software framework that could enable rapid development and deployment of innovative privacy-aware and multi-tenant applications for the emerging Internet of Things.

7.2.3 Data Poisoning

Data poisoning risks are becoming increasingly challenging as the amount of data grows rapidly. New techniques are needed to detect and measure the data poisoning risks to massive datasets. From this angle, in the long term, we are particularly interested in utilizing data mining and machine learning techniques to effectively detect and avoid the data poisoning in next-generation systems 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 issues.

Bibliography

[1] A Microscope on Microservices. http://goo.gl/1BXHRH.

[2] Advanced Message Queuing Protocol. https://www.amqp.org/.

[3] Arq. https://www.haystacksoftware.com/arq/.

[4] Artifactory. https://www.jfrog.com/confluence/display/RTF/Docker+Registry.

[5] Bottle: Python Web Framework. https://github.com/bottlepy/bottle.

[6] Ceph vs Swift. http://goo.gl/rtvrvg.

[7] Control Groups. https://goo.gl/KkCXAR.

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

[9] CoreOS. https://coreos.com/.

[10] IBM CPLEX Optimizer. http://goo.gl/BA95mC.

[11] OpenTSDB as a metering storage for OpenStack Telemetry. https://goo.gl/3I6L6J.

[12] What is Docker, . https://www.docker.com/what-docker.

[13] Dockerhub, . https://hub.docker.com.

[14] Docker run preferences. https://goo.gl/SoF9Pc, .

[15] Docker-Registry. https://github.com/docker/docker-registry.

[16] Docker container image for Swift object server. https://hub.docker.com/r/alivt/swift-object.

[17] Docker container image for Swift proxy sever. https://hub.docker.com/r/alivt/swift-proxy.

[18] dxf. https://github.com/davedoesdev/dxf.

[19] Netflix Dynomite. https://github.com/Netflix/dynomite.

[20] Amazon ElastiCache. https://aws.amazon.com/elasticache/.

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

[22] CoreOS etcd. https://github.com/coreos/etcd.

[23] Google Container Registry. https://cloud.google.com/container-registry/.

[24] gRPC. https://grpc.io/.

[25] Project Harbor. https://github.com/vmware/harbor.

[26] Apache HBase. http://hbase.apache.org/.

[27] HelloBench. https://github.com/Tintri/hello-bench.

[28] Cloud Backup with HP Public Cloud. http://goo.gl/43bC5W.

[29] Writing rules for the state correlation engine, . http://goo.gl/Z5McMT.

[30] Tivoli Application Dependency Discovery Manager, . http://goo.gl/tO8a4b.

[31] IBM Cloud Container Registry, . https://console.bluemix.net/docs/services/Registry/index.html.

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

[33] libMemcached, . http://libmemcached.org/libMemcached.html.

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

[35] Linux Traffic Control. http://goo.gl/E5aQdq.

[36] Manhattan, our real-time, multi-tenant distributed database for Twitter scale. goo.gl/7EThfo.

[37] Memcached. https://memcached.org/.

[38] Microservices and Docker containers, . goo.gl/UrVPdU.

[39] Microservices Architecture, Containers and Docker., . goo.gl/jsQ1sL.

[40] MongoDB. https://www.mongodb.com/.

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

[42] Quay.io. https://quay.io/.

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

[44] Redis clients, . https://redis.io/clients.

[45] Redis Cluster Spec, . http://redis.io/topics/cluster-spec.

[46] Ceilometer Quickstart, . http://goo.gl/LEYBiM.

[47] Redis, . https://redis.io/.

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

[49] repcached - add data replication feature to memcached 1.2.x. http://repcached.lab.klab.org/.

[50] Rethinking Ceilometer metric storage with Gnocchi: Time-series as a Service. https://goo.gl/68jXaO.

[51] The R Project for Statistical Computing. http://www.r-project.org/.

[52] Amazon S3. https://aws.amazon.com/s3.

[53] Ceilometer Samples and Statistics. http://goo.gl/u2hBKe.

[54] Symbolic Aggregate approXimation. http://goo.gl/30X8f1.

[55] SSDB. https://github.com/ideawu/ssdb.

[56] OpenStack Swift, . https://docs.openstack.org/swift/.

[57] Swfit Storage Policies. http://goo.gl/hRrySo, .

[58] The SYSSTAT utilities. http://goo.gl/21243V.

[59] Volt git repo. https://github.com/VoltFramework/volt.

[60] Vsync. https://vsync.codeplex.com/.

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

[62] 451 Research. Application Containers Will Be a $2.7Bn Market by 2020. http://bit.ly/2uryjDI.

[63] Manoj Agarwal, Manish Gupta, Vijay Mann, Narendran Sachindran, Nikos Anerousis, and Lily Mummert. Problem determination in enterprise middleware systems using change point correlation of time series data. In *IEEE/IFIP NOMS*, 2006.

[64] Marcos K Aguilera, Joshua B Leners, and Michael Walfish. Yesquel: scalable sql storage for web applications. In *ACM SOSP'15*.

[65] Marcelo Amaral, Jorda Polo, David Carrera, Iqbal Mohomed, Merve Unuvar, and Malgorzata Steinder. Performance evaluation of microservices architectures using containers. In *IEEE NCA*, 2015.

[66] David G. Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. Fawn: A fast array of wimpy nodes. In *ACM SOSP*, 2009.

[67] 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.

[68] Ali Anwar, Yue Cheng, Hai Huang, and Ali R. Butt. Clusteron: Building highly configurable and reusable clustered data services using simple data nodes. In *USENIX HotStorage'16*.

[69] Ali Anwar, Andrzej Kochut Anca Sailer, Charles O. Schulz, Alla Segal, and Ali R. Butt. Scalable metering for an affordable it cloud service management. In *Proceedings of the IEEE International Conference on Cloud Engineering*, IC2E '15, Tempe, AZ, March 2015.

[70] Ali Anwar, Yue Cheng, Aayush Gupta, and Ali R Butt. Taming the cloud object storage with mos. In *PDSW*. ACM, 2015.

[71] Ali Anwar, Yue Cheng, Aayush Gupta, and Ali R. Butt. Taming the cloud object storage with mos. In *ACM PDSW*, 2015.

[72] Ali Anwar, Anca Sailer, Andrzej Kochut, and Ali R Butt. Anatomy of cloud monitoring and metering: A case study and open problems. In *Proceedings of the 6th Asia-Pacific Workshop on Systems*, page 6. ACM, 2015.

[73] Ali Anwar, Anca Sailer, Andrzej Kochut, Charles O. Schulz, Segal Alla, and Ali R. Butt. Cost-aware cloud metering with scalable service management infrastructure. In *IEEE CLOUD*, 2015.

[74] Ali Anwar, Anca Sailer, Andrzej Kochut, Charles O. Schulz, Segal Alla, and Ali R. Butt. Scalable metering for an affordable it cloud service management. In *IEEE IC2E*, 2015.

[75] Ali Anwar, Anca Sailer, Andrzej Kochut, Charles O Schulz, Alla Segal, and Ali R Butt. Cost-aware cloud metering with scalable service management infrastructure. In *Proceedings of the IEEE 2015 International Conference on Cloud Computing (IEEE Cloud)*, New York, 2015.

[76] Ali Anwar, Yue Cheng, Aayush Gupta, and Ali R. Butt. Mos: Workload-aware elasticity for cloud object stores. In *ACM HPDC*, 2016.

[77] Ali Anwar, Mohamed Mohamed, Vasily Tarasov, Michael Littley, Lukas Rupprecht, Yue Cheng, Nannan Zhao, Dimitrios Skourtis, Amit S Warke, Heiko Ludwig, et al. Improving docker registry design based on production workload analysis. In *16th USENIX Conference on File and Storage Technologies*, page 265, 2018.

[78] Ayesha Anwar, KR Krish, and Ali R Butt. On the use of microservers in supporting hadoop applications. In *CLUSTER*. IEEE, 2014.

[79] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *ACM SIGMETRICS*, 2012.

[80] Doug Beaver, Sanjeev Kumar, Harry C Li, Jason Sobel, Peter Vajgel, et al. Finding a needle in haystack: Facebook's photo storage. In *USENIX OSDI*, 2010.

[81] Nicolas Bonvin, Thanasis G Papaioannou, and Karl Aberer. A self-organized, fault-tolerant and scalable replication scheme for cloud storage. In *SOCC*. ACM, 2010.

[82] Andre Brinkmann, Christoph Fiehe, Anna Litvina, Ingo Luck, Lars Nagel, Krishnaprasad Narayanan, Florian Ostermair, and Wolfgang Thronicke. Scalable monitoring system for clouds. In *IEEE/ACM UCC*, 2013.

[83] Ali R. Butt, Chris Gniady, and Y. Charlie Hu. The performance impact of kernel prefetching on buffer cache replacement algorithms. In *ACM SIGMETRICS*, 2005.

[84] Ali R. Butt, Ali Anwar, and Yue Cheng. Sdn helps big data to optimize storage. In *Big Data and Software Defined Networks*. IET Digital Library, 2018.

[85] Claudia Canali and Riccardo Lancellotti. Automatic virtual machine clustering based on bhattacharyya distance for multi-cloud systems. In *International workshop on Multi-cloud applications and federated clouds*. ACM, 2013.

[86] Pei Cao, Edward W. Felten, Anna R. Karlin, and Kai Li. Implementation and performance of integrated application-controlled file caching, prefetching, and disk scheduling. *ACM Trans. Comput. Syst.*, 14(4):311–343, November 1996. ISSN 0734-2071. doi: 10.1145/235543.235544. URL http://doi.acm.org/10.1145/235543.235544.

[87] Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: An engineering perspective. ACM PODC, 2007.

[88] Chen Chen, Petros Maniatis, Adrian Perrig, Amit Vasudevan, and Vyas Sekar. Towards verifiable resource accounting for outsourced computation. In *ACM SIGPLAN Notices*, 2013.

[89] Jianjun Chen, Chris Douglas, Michi Mutsuzaki, Patrick Quaid, Raghu Ramakrishnan, Sriram Rao, and Russell Sears. Walnut: a unified cloud object store. In *SIGMOD*. ACM, 2012.

[90] Ming Chen, Dean Hildebrand, Geoff Kuenning, Soujanya Shankaranarayana, Bharat Singh, and Erez Zadok. Newer is sometimes better: An evaluation of nfsv4.1. In *ACM SIGMETRICS*, 2015.

[91] Yanpei Chen, Kiran Srinivasan, Garth Goodson, and Randy Katz. Design implications for enterprise storage systems via multi-dimensional trace analysis. In *ACM SOSP*, 2011.

[92] Yue Cheng. Workload-aware efficient storage systems. In *Doctoral dissertation*. Virginia Tech Library, 2017.

[93] Yue Cheng, Aayush Gupta, Anna Povzner, and Ali R. Butt. High performance in-memory caching through flexible fine-grained services. In *ACM SOCC*, 2013.

[94] Yue Cheng, Aayush Gupta, and Ali R. Butt. An in-memory object caching framework with adaptive load balancing. In *ACM EuroSys*, 2015.

[95] Yue Cheng, M. Safdar Iqbal, Aayush Gupta, and Ali R. Butt. Cast: Tiering storage for data analytics in the cloud. In *HPDC*. ACM, 2015. ISBN 978-1-4503-3550-8. doi: 10.1145/2749246.2749252. URL http://doi.acm.org/10.1145/2749246.2749252.

[96] 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, 2015. ACM. ISBN 978-1-4503-3550-8. doi: 10.1145/2749246.2749252. URL http://doi.acm.org/10.1145/2749246.2749252.

[97] Yue Cheng, M Safdar Iqbal, Aayush Gupta, and Ali R Butt. Pricing games for hybrid object stores in the cloud: provider vs. tenant. In *USENIX HotCloud*, 2015.

[98] 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, 2015. USENIX Association. URL https://www.usenix.org/conference/hotcloud15/workshop-program/presentation/cheng.

[99] Yue Cheng, Fred Douglis, Philip Shilane, 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)*, pages 379–392, Denver, CO, 2016. USENIX Association. ISBN 978-1-931971-30-0. URL https://www.usenix.org/conference/atc16/technical-sessions/presentation/cheng.

[100] 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.

[101] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *ACM SoCC '10*.

[102] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *VLDB Endow.*, 2008.

[103] Francisco Cruz, Francisco Maia, Miguel Matos, Rui Oliveira, João Paulo, José Pereira, and Ricardo Vilaça. Met: workload aware elasticity for nosql. In *EuroSys*. ACM, 2013.

[104] Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. Elastras: An elastic transactional data store in the cloud. *USENIX HotCloud*, 2009.

[105] 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.

[106] Pantazis Deligiannis, Matt McCutchen, Paul Thomson, Shuo Chen, Alastair F. Donaldson, John Erickson, Cheng Huang, Akash Lal, Rashmi Mudduluru, Shaz Qadeer, and Wolfram Schulte. Uncovering bugs in distributed storage systems during testing (not in production!). In *USENIX FAST '16*.

[107] Christina Delimitrou and Christos Kozyrakis. Paragon: Qos-aware scheduling for heterogeneous datacenters. In *ASPLOS*. ACM, 2013. ISBN 978-1-4503-1870-9. doi: 10.1145/2451116.2451125. URL http://doi.acm.org/10.1145/2451116.2451125.

[108] Dell EMC. Improving Copy-on-Write Performance in Container Storage Drivers. https://www.snia.org/sites/default/files/SDC/2016/presentations/capacity_optimization/FrankZaho_Improving_COW_Performance_ContainerStorage_Drivers-Final-2.pdf.

[109] Robert Escriva, Bernard Wong, and Emin Gün Sirer. Hyperdex: A distributed, searchable key-value store. In *ACM SIGCOMM '12*.

[110] Bin Fan, Hyeontaek Lim, David G. Andersen, and Michael Kaminsky. Small cache, big effect: Provable load balancing for randomly partitioned cluster services. In *ACM SOCC '11*.

[111] Wes Felter, Alexandre Ferreira, Ram Rajamony, and Juan Rubio. An updated performance comparison of virtual machines and linux containers. *technology*, 2014.

[112] Wes Felter, Alexandre Ferreira, Ram Rajamony, and Juan Rubio. An Updated Performance Comparison of Virtual Machines and Linux Containers. In *IEEE ISPASS*, 2015.

[113] Mainak Ghosh, Wenting Wang, Gopalakrishna Holla, and Indranil Gupta. Morphus: Supporting online reconfigurations in sharded nosql systems. *IEEE Transactions on Emerging Topics in Computing*, 2015.

[114] Chris Gniady, Ali R. Butt, and Y. Charlie Hu. Program-counter-based pattern classification in buffer caching. In *USENIX OSDI*, 2004.

[115] Hugh Greenberg, John Bent, and Gary Grider. Mdhim: A parallel key/value framework for hpc. In *USENIX HotStorage*, 2015.

[116] Haryadi S. Gunawi, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, Thanh Do, Jeffry Adityatama, Kurnia J. Eliazar, Agung Laksono, Jeffrey F. Lukman, Vincentius Martin, and Anang D. Satria. What bugs live in the cloud? a study of 3000+ issues in cloud systems. In *ACM SOCC*, 2014.

[117] Tyler Harter, Brandon Salmon, Rose Liu, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Slacker: Fast Distribution with Lazy Docker Containers. In *USENIX FAST*, 2016.

[118] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *USENIX NSDI*, 2011.

[119] Qi Huang, Ken Birman, Robbert van Renesse, Wyatt Lloyd, Sanjeev Kumar, and Harry C. Li. An analysis of facebook photo caching. In *ACM SOSP*, 2013.

[120] Galen C Hunt, Michael L Scott, et al. The coign automatic distributed partitioning system. In *USENIX OSDI'99*, .

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

[122] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX USENIX*, 2010. URL http://dl.acm.org/citation.cfm?id=1855840.1855851.

[123] Jinho Hwang, K. K. Ramakrishnan, and Timothy Wood. Netvm: High performance and flexible networking using virtualization on commodity platforms. In *USENIX NSDI '14*.

[124] Ravi Iyer, Ramesh Illikkal, Li Zhao, Don Newell, and Jaideep Moses. Virtual platform architectures for resource metering in datacenters. *ACM SIGMETRICS*, 2009.

[125] Xuxian Jiang and Xinyuan Wang. "out-of-the-box" monitoring of vm-based high-interaction honeypots. In *Recent Advances in Intrusion Detection*, pages 198–218. Springer, 2007.

[126] Wang Kangjin, Yang Yong, Li Ying, Luo Hanmei, and Ma Lin. Fid: A faster image distribution system for docker platform. In *IEEE AMLCS*, 2017.

[127] Srinivas Kashyap, Jeyashankher Ramamirtham, Rajeev Rastogi, and Pushpraj Shukla. Efficient constraint monitoring using adaptive thresholds. In *IEEE ICDE*, 2008.

[128] KR Krish, Ali Anwar, and Ali R Butt. [phi] sched: A heterogeneity-aware hadoop workflow scheduler. In *MASCOTS*. IEEE, 2014.

[129] KR Krish, Ayesha Anwar, and Ali R Butt. hats: A heterogeneity-aware tiered storage for hadoop. In *CCGrid*. IEEE, 2014.

[130] Charles W. Krueger. Software reuse. *ACM Comput. Surv.*, 24(2):131–183, June 1992. ISSN 0360-0300.

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

[132] Leslie Lamport. Paxos makde simple. In *ACM SIGACT News '01.*

[133] Gunho Lee, Byung-Gon Chun, and Randy H Katz. Heterogeneity-aware resource allocation and scheduling in the cloud. In *USENIX HotCloud*, 2011.

[134] Tanakorn Leesatapornwongsa, Jeffrey F. Lukman, Shan Lu, and Haryadi S. Gunawi. Taxdc: A taxonomy of non-deterministic concurrency bugs in datacenter distributed systems. ACM ASPLOS '16.

[135] Tanakorn Leesatapornwongsa, Mingzhe Hao, Pallavi Joshi, Jeffrey F. Lukman, and Haryadi S. Gunawi. Samc: Semantic-aware model checking for fast discovery of deep bugs in cloud systems. In *USENIX OSDI*, 2014.

[136] Mingju Li, Elizabeth Varki, Swapnil Bhatia, and Arif Merchant. Tap: Table-based prefetching for storage caches. In *USENIX FAST*, 2008.

[137] Xiaozhou Li, Raghav Sethi, Michael Kaminsky, David G. Andersen, and Michael J. Freedman. Be fast, cheap and in control with switchkv. In *USENIX NSDI '16.*

[138] J. Liao, F. Trahay, G. Xiao, L. Li, and Y. Ishikawa. Performing initiative data prefetching in distributed file systems for cloud computing. *IEEE Transactions on Cloud Computing*, 5(3):550–562, July 2017.

[139] Harold C Lim, Shivnath Babu, and Jeffrey S Chase. Automated control for elastic storage. In *ICAC*. ACM, 2010.

[140] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. Mica: A holistic approach to fast in-memory key-value storage. In *USENIX NSDI'14*.

[141] Jessica Lin, Eamonn Keogh, Li Wei, and Stefano Lonardi. Experiencing sax: a novel symbolic representation of time series. *Data Mining and knowledge discovery*, 15(2): 107–144, 2007.

[142] Mei Liu and Xuhua Ding. On trustworthiness of cpu usage metering and accounting. In *IEEE ICDCSW*, 2010.

[143] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't settle for eventual: Scalable causal consistency for wide-area storage with cops. In *ACM SOSP '11*.

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

[145] J. Mars, Lingjia Tang, and R. Hundt. Heterogeneity in "homogeneous" warehouse-scale computers: A performance opportunity. *CAL*, 2011. ISSN 1556-6056. doi: 10.1109/L-CA.2011.14.

[146] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, March 2008. ISSN 0146-4833.

[147] Paul B Menage. Adding Generic Process Containers to the Linux Kernel. In *Linux Symposium*, 2007.

[148] Shicong Meng, Arun K Iyengar, Isabelle M Rouvellou, Ling Liu, Kisung Lee, Balaji Palanisamy, and Yuzhe Tang. Reliable state monitoring in cloud datacenters. In *IEEE CLOUD*, 2012.

[149] Shicong Meng, Arun K Iyengar, Isabelle M Rouvellou, and Ling Liu. Volley: Violation likelihood based state monitoring for datacenters. In *IEEE ICDCS*, 2013.

[150] Senthil Nathan, Rahul Ghosh, Tridib Mukherjee, and Krishnaprasad Narayanan. CoMICon: A Co-Operative Management System for Docker Container Images. In *IEEE IC2E*, 2017.

[151] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *USENIX ATC 14*.

[152] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed prefetching and caching. In *ACM SOSP*, 1995.

[153] Amar Phanishayee, David G. Andersen, Himabindu Pucha, Anna Povzner, and Wendy Belluomini. Flex-KV: Enabling high-performance and flexible KV systems. In *Proc. Workshop on Management of Big Data Systems*, September 2012.

[154] Wolfgang Richter, Canturk Isci, Benjamin Gilbert, Jan Harkes, Vasanth Bala, and Mahadev Satyanarayanan. Agentless cloud-wide streaming of guest file system updates. In *IEEE IC2E*, 2014.

[155] Michael A Sevilla, Noah Watkins, Carlos Maltzahn, Ike Nassi, Scott A Brandt, Sage A Weil, Greg Farnum, and Sam Fineberg. Mantle: a programmable metadata load balancer for the ceph file system. In *SC*. ACM, 2015.

[156] Pekka Siirtola, Heli Koskimäki, Ville Huikari, Perttu Laurinen, and Juha Röning. Improving the classification accuracy of streaming data using sax similarity features. *Pattern Recognition Letters*, 32(13):1659–1668, 2011.

[157] Hyogi Sim, Youngjae Kim, Sudharshan S Vazhkudai, Devesh Tiwari, Ali Anwar, Ali R Butt, and Lavanya Ramakrishnan. Analyzethis: an analysis workflow-aware storage system. In *High Performance Computing, Networking, Storage and Analysis, 2015 SC-International Conference for*, pages 1–12. IEEE, 2015.

[158] Stephen Soltesz, Herbert Pötzl, Marc E. Fiuczynski, Andy Bavier, and Larry Peterson. Container-based Operating System Virtualization: A Scalable, High-performance Alternative to Hypervisors. In *ACM EuroSys*, 2007.

[159] Yusuke Tanimura, Seiya Yanagita, and Takahiro Hamanishi. A high performance, qos-enabled, s3-based object store. In *CCGrid*. IEEE, 2014.

[160] Vasily Tarasov, Lukas Rupprecht, Dimitris Skourtis, Amit Warke, Dean Hildebrand, Mohamed Mohamed, Nagapramod Mandagere, Wenji Li, Raju Rangaswami, and Ming Zhao. In search of the ideal storage configuration for Docker containers. In *IEEE AMLCS*, 2017.

[161] Douglas B. Terry, Vijayan Prabhakaran, Ramakrishna Kotla, Mahesh Balakrishnan, Marcos K. Aguilera, and Hussam Abu-Libdeh. Consistency-based service level agreements for cloud storage. In *ACM SOSP'13*.

[162] Eno Thereska, Hitesh Ballani, Greg O'Shea, Thomas Karagiannis, Antony Rowstron, Tom Talpey, Richard Black, and Timothy Zhu. Ioflow: A software-defined storage architecture. In *SOSP*. ACM, 2013.

[163] Beth Trushkowsky, Peter Bodík, Armando Fox, Michael J Franklin, Michael I Jordan, and David A Patterson. The scads director: Scaling a distributed storage system under stringent performance requirements. In *USENIX FAST*, 2011.

[164] Luis M Vaquero, Luis Rodero-Merino, Juan Caceres, and Maik Lindner. A break in the clouds: towards a cloud definition. *ACM SIGCOMM Computer Communication Review*, 2008.

[165] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *USENIX OSDI '06*.

[166] Sage A Weil, Scott A Brandt, Ethan L Miller, Darrell DE Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *USENIX OSDI*, 2006.

[167] S. P. Vander Wiel and D. J. Lilja. When caches aren't enough: data prefetching techniques. *Computer*, 30(7):23–30, Jul 1997.

[168] Irene Zhang, Naveen Kr. Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan R. K. Ports. Building consistent transactions with inconsistent replication. In *ACM SOSP'15*, .

[169] W. Zhang, T. Wood, and J. Hwang. Netkv: Scalable, self-managing, load balancing as a network function. In *ICAC '16*, .

[170] Wei Zhang, Jinho Hwang, Timothy Wood, K.K. Ramakrishnan, and Howie Huang. Load balancing of heterogeneous workloads in memcached clusters. In *Feedback Computing 14*, .

[171] Zhe Zhang, Amit Kulkarni, Xiaosong Ma, and Yuanyuan Zhou. Memory resource allocation for file system prefetching: From a supply chain management perspective. In *ACM EuroSys*, 2009.

[172] Qing Zheng, Haopeng Chen, Yaguang Wang, Jiangang Duan, and Zhiteng Huang. Cosbench: A benchmark tool for cloud object storage services. In *CLOUD*. IEEE, 2012.