FAST, ELASTIC STORAGE FOR THE CLOUD

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

Ana Klimovic
May 2019

This dissertation is online at: http://purl.stanford.edu/mn535mm5645

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

Christos Kozyrakis, Primary Adviser

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

Mark Horowitz

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

Matei Zaharia

Approved for the Stanford University Committee on Graduate Studies.

Patricia J. Gumport, Vice Provost for Graduate Education

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

Abstract

Cloud computing promises high performance, cost-efficiency, and elasticity — three essential goals when processing exponentially growing datasets. To meet these goals, cloud platforms must provide each application with the right amount and balance of compute and fast storage resources (e.g., Flash storage). However, providing the right resources to applications is difficult for several reasons: server machines have a fixed ratio of compute and storage resources, remote access to fast storage leads to significant performance and cost overheads, and storage requirements vary significantly over time and across applications.

This dissertation focuses on how to build high performance, cost-effective, and easy-to-use cloud storage systems. We address two critical challenges. The first challenge is providing *fast, predictable access to remote data* on modern storage devices, so that the balance of compute and storage allocation is not limited by the physical characteristics of server hardware. Our insight is to disaggregate Flash storage from compute resources while maintaining high quality of service (QoS) with a novel software stack for network and storage processing. The second challenge is *intelligent allocation of storage resources*. We discuss how to automate storage resource management by rightsizing resource allocations based on application characteristics, learned by user hints or by training a machine learning model on past performance data

To address the first challenge of fast, predictable access to remote data, we present ReFlex, a custom network-storage operating system. ReFlex improves resource allocation flexibility by enabling applications to access remote Flash over commodity cloud networks without sacrificing the performance and predictability achieved with local Flash. By closely integrating networking and storage processing, the software-based system serves up to 850,000 I/O operations per second per core over TCP/IP networking, while adding only 21 μs latency over direct access to local Flash. Compared to using traditional Linux I/O libraries (`libevent` and `libaio`) for remote access to Flash, ReFlex achieves an 11× improvement in throughput per core and a 3× improvement in latency. The system can also enforce tail latency and throughput service-level objectives for thousands of remote clients sharing a remote Flash device using a novel, QoS-aware I/O scheduler.

Though ReFlex gives us the flexibility to customize the balance of compute and Flash storage

resource allocations for each application, selecting the right resource allocations to optimize performance and cost efficiency is challenging. We address the challenge of intelligent resource allocation in a new storage service called Pocket, which builds on top of ReFlex. Pocket enables an emerging class of applications called serverless analytics to run efficiently in the cloud by providing fast, elastic storage for intermediate data sharing between tasks. The distributed data store dynamically rightsizes resources across multiple dimensions (CPU cores, network bandwidth, and storage capacity) and leverages multiple storage technologies to minimize cost while ensuring applications are not bottlenecked on I/O. The system leverages optional hints, which can be provided by application frameworks or users, to determine a job's latency, throughput, and capacity requirements. Pocket reduces the average time that serverless tasks in a video analytics application spend on ephemeral data I/O by over 4× compared to Amazon S3, a popular object storage service. Pocket offers applications similar performance to a popular memory-based data store, Redis, while rightsizing and autoscaling storage resources to save close to 60% in cost.

Finally, we discuss the potential of using machine learning to automate intelligent resource allocation decisions and replace the hints and heuristics commonly used in today's resource management systems. We built a system called Selecta that recommends a near-optimal resource allocation for a target application by predicting the applications execution time across a variety of virtual machine and storage configurations. The system learns by leveraging performance data collected for applications that were previously run across a variety configurations. Using a collaborative filtering technique to uncover latent similarities across jobs and resource configurations, Selecta makes fast, accurate recommendations with sparse training data. For example, Selecta has a 94% probability of recommending a configuration that performs within 10% of the true highest performance configuration. Our evaluation of Selecta for data analytics jobs across various compute and storage configurations also revealed several key insights to guide the design of future storage systems, such as the need for fine-grain allocation of storage capacity and bandwidth.

Acknowledgments

Pursuing a PhD has been an incredible journey and I am deeply grateful for the support I have received from my mentors, family, and friends throughout.

First and foremost, I would like to thank my advisor, Christos Kozyrakis, for guiding me towards fruitful research directions while giving me the freedom to find my own path. Christos has inspired me in many ways, in particular with his exceptional ability to recognize important research challenges and build a positive research group culture. I have enjoyed and learned so much from our discussions about research and life in academia.

I would like to thank Mark Horowitz for his excellent advice, which helped me find my way at critical crossroads in the PhD journey. Mark knows exactly how to ask the right questions and give honest, wise advice. I am grateful to Matei Zaharia for his enthusiastic guidance on research and career paths, which helped shape the projects I pursued in my last two years of graduate school. Thank you to Mendel Rosenblum and Mykel Kochenderfer for their support and for being on my dissertation committee.

I am fortunate to have collaborated with brilliant colleagues in academia and industry on the work described in this dissertation. Thank you to Heiner Litz, Patrick Stuedi, Yawen Wang, Adam Belay, Eno Thereska, Binu John, Animesh Trivedi, and Jonas Pfefferle for their valuable contributions to this work. I am proud of what we have accomplished together.

Thank you to the MAST research group whose members have made my day-to-day research experience a happy one: Christina Delimitrou, Yawen Wang, Kostis Kaffes, Mingyu Gao, Heiner Litz, Grant Ayers, David Lo, Felipe Munera, Raghu Prabhakar, Adam Belay, Sam Grossman, Franky Romero, Qian Li, Tim Chong, Neeraja Yadwadkar, Mark Zhao, Geet Sethi, Greg Kehoe, and Youngjin Cho. Thank you also to all the members of the Stanford Platform Lab.

I am thankful for the financial support I received from the Stanford Graduate Fellowship and Microsoft Research PhD Fellowship, along with the entrepreneurial training I gained in the Accel Scholars program. I am grateful to the Capacity Engineering team at Facebook for hosting me for a memorable summer research internship that inspired the topic of this dissertation. Thank you also to the Systems and Networking group that I interned with at Microsoft Research in Cambridge, UK for the valuable mentorship.

Many friends contributed to making my PhD experience fulfilling. I would like to thank Alysson for being a wonderful friend and a constant source of positive energy and encouragement. Thank you to Maryia, Lisa, Sylvia, Victor, David, Amory, Bushra, Joyce, Adrien, Ognjen, Djordje, and many others for the great memories together at Stanford. I am incredibly grateful to have Stephanie, Aleksandra, and Nataša as such close friends, even when we are thousands of miles apart.

I am forever grateful to my family and in particular to my parents, Erna and Milorad. Words cannot express how valuable their encouragement and advice over the years has been. I would not be who or where I am today without their love and support.

A very special thank you goes to Taško Olevski. I am incredibly fortunate to have such a loving, caring, and supportive partner. Thank you for bringing so much joy to my life.

Contents

List of Tables

List of Figures

Previously Published Material

Chapter 2 includes content from two previous publications [121, 124].

- Ana Klimovic, Christos Kozyrakis, Eno Thereska, Binu John, and Sanjeev Kumar. *Flash storage disaggregation.* In the Proceedings of European Conference on Computer Systems (EuroSys), 2016.
- Ana Klimovic, Yawen Wang, Christos Kozyrakis, Patrick Stuedi, Jonas Pfefferle, and Animesh Trivedi. *Understanding ephemeral storage for serverless analytics.* In the Proceedings of the USENIX Annual Technical Conference (ATC), 2018.

Chapter 3 revises a previous publication [122].

- Ana Klimovic, Heiner Litz, and Christos Kozyrakis. *ReFlex: Remote Flash ≈ Local Flash.* In the Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2017.

Chapter 4 revises a previous publication [125].

- Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. *Pocket: Elastic ephemeral storage for serverless analytics.* In the Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI), 2018.

Chapter 5 revises a previous publication [123].

- Ana Klimovic, Heiner Litz, and Christos Kozyrakis. *Selecta: Heterogeneous cloud storage configuration for data analytics.* In the Proceedings of the USENIX Annual Technical Conference (ATC), 2018.

Chapter 1

Introduction

The amount of data that we are generating and analyzing is growing exponentially. A recent study estimates that the amount of data being recorded is doubling every two to three years [129]. Furthermore, our desire to extract knowledge from the data that we collect, with techniques such as machine learning, means that an increasing amount of data is being actively processed. To store and process these growing datasets, users are commonly turning to cloud computing platforms. This is because cloud computing promises high elasticity, cost-efficiency, scalability, and performance [30]. However, these benefits of cloud computing are only achieved if each application receives the type and amount of storage and compute resource it needs.

Provisioning cloud resources to achieve performance and cost efficiency is difficult for several reasons. The compute and storage resource requirements of applications vary dramatically over time and across jobs [121, 29]. Furthermore, compute and storage resource requirements are not necessarily correlated [137, 194]. For example, a change to application code can substantially improve CPU efficiency, thus decreasing compute requirements, while storage resource demands continue to increase due to increasing application load. Application resource requirements are only becoming more dynamic. For example, with the increasing popularity of elastic computing services such as "serverless computing", users can simultaneously launch thousands of short-lived tasks with fine-grain resource scaling and billing.

While application requirements vary dynamically, the servers hosting applications in cloud facilities have fixed ratios of compute and storage resources. For instance, the number of CPU cores and storage device(s) on a node is fixed when a server is built and deployed. Cloud operators typically limit variety of server configurations deployed in a datacenter as high heterogeneity would make maintenance intractable at the scale of tens of thousands of servers [74]. The mismatch between the dynamic nature of application resource requirements and the static set of resources available on datacenter servers leads to a lack of flexibility for resource allocation. For example, a cloud application may require 1 TB of Flash storage capacity and 50 CPU cores for request processing, but Flash

storage servers in the cloud facility may be deployed with a ratio of 1 TB of Flash storage to 8 CPU cores. As a result, resources like Flash storage are commonly over-provisioned and underutilized. Over-provisioning resources significantly increases cost, particularly since the server equipment is the highest contributor to the total cost of ownership for warehouse-scale datacenters [97]. With tens of thousands of servers per datacenter facility consuming tens of megawatts of power, resource underutilization at this scale is extremely wasteful.

Fortunately, there are promising opportunities to improve datacenter resource utilization. For example, by sharing storage resources over the network and enabling flexible allocation of storage and compute resources, customized to the needs of individual applications. However, we must improve resource utilization while maintaining high performance. Hence, the overarching goal is to design *resource-efficient* cloud storage systems, meaning systems that satisfy application service level objectives (SLOs) while provisioning the minimum required resources for each job.

To improve the resource efficiency of existing cloud storage systems, we have two main requirements. First, we need flexible resource allocation. Instead of allocating resources in units of physical servers with fixed compute and storage capacities, we would like to allocate compute and storage from independent resource pools so that each application can get exactly the right amount and type of resources it needs. Second, we need intelligent controllers to make performance and cost effective resource allocation decisions at large scale. Automating storage resource allocation decisions is necessary to keep up with the growing number of cloud applications and their increasingly dynamic resource requirements, thanks to the elasticity of new serverless cloud computing platforms.

1.1 Contributions

The key question this dissertation explores is *how should we build resource-efficient and high performance storage systems for diverse, elastic applications in the cloud?*. We address this question using two main insights. The first insight is to decouple storage from compute resources to improve resource allocation flexibility. A major challenge is enabling fast, predictable access to modern Flash storage devices over commodity cloud networks. While networking hardware enables low latency and high throughput access across servers in the whole datacenter, traditional operating system software for network storage request processing is currently a bottleneck [38, 162]. The second insight is to make intelligent resource allocation decisions at scale by learning application characteristics, either from hints or machine learning models trained on prior performance data. We design a controller that interprets hints provided through a novel storage API to allocate and autoscale resources efficiently for each job. To eliminate the burden of providing hints, we explore how to autonomously learn near-optimal resource allocations based on sparse data collected from prior application runs.

Using these two key insights, we design and implement three systems that collectively improve cloud storage resource efficiency and performance. Below, we provide a brief overview of each system.

ReFlex: Fast, Predictable Access to Remote Flash

Storage disaggregation addresses the challenge of imbalanced resource requirements and the resulting overprovisioning of datacenter resources by physically decoupling compute and storage resources, such that datacenter operators can more easily customize their infrastructure to maximize the performance-per-dollar for target workloads. In a disaggregated cloud environment, applications remotely access storage devices that have spare capacity and bandwidth, regardless of their physical location in a cloud facility. This approach is already common for disk storage as network overheads are small compared to a disk's millisecond access latency and low hundreds of IOPS [25]. However, disaggregating modern Flash storage devices presents several challenges. Traditional operating systems introduce high overhead as they were originally designed with the assumption of slow network and storage devices. Furthermore, since write operations take significantly longer than read operations on Flash technology, performance isolation between read and write requests from different tenants sharing a Flash device is important for quality of service.

ReFlex is a software system that enables applications to access remote Flash storage with nearly identical performance to accessing local Flash. ReFlex runs on remote Flash storage servers and bypasses traditional operating system software for network and storage processing. By efficiently integrating network and storage processing, ReFlex serves up to 850,000 storage requests per second per CPU core on an Intel Xeon server. This is an 11× improvement compared to a Linux server implemented with standard I/O libraries. ReFlex provides throughput and tail latency performance guarantees for up to thousands of clients sharing a Flash server with a novel I/O scheduler that takes into account Flash device characteristics and client performance objectives. In contrast to approaches that rely on specialized networking hardware to achieve high performance, ReFlex is a purely software-based solution that works on commodity hardware in public clouds.

ReFlex is highly adoptable as it requires no changes to the host operating system. The system is open-source and is already having impact. ReFlex serves as a fast Flash storage tier on public clouds for the Apache Crail distributed data store, which enables efficient data sharing in widely used data processing frameworks, such as Spark and TensorFlow [197, 4]. Broadcom is also integrating the ReFlex I/O scheduler on a system on chip platform to provide quality of service guarantees to tenants sharing a Flash device over the network.

Pocket: Fast, Elastic Storage for Ephemeral Data

While ReFlex provides a fast data plane for access to remote blocks of data, a complete cloud storage solution also requires a metadata plane to manage how application objects are stored as data blocks across storage nodes and an intelligent control plane to allocate storage resources in a cluster. We introduce a new system, Pocket, which adds metadata management and intelligent resource allocation on top of ReFlex's fast data plane. We design Pocket to enable an emerging class of cloud applications, called serverless analytics, to run efficiently in the cloud by providing fast,

elastic storage for ephemeral data sharing between tasks.

Serverless computing is a fully managed compute service that is becoming increasingly popular, enabling users to quickly launch thousands of light-weight tasks in the cloud, as opposed to entire virtual machines. Though initially designed for simple applications that execute a single task in response to an event, several emerging application frameworks are leveraging serverless computing to exploit massive task parallelism and achieve near real-time performance for more complex jobs, such as data analytics. A key challenge for serverless analytics is efficiently managing temporary data that tasks generate and exchange between different stages of execution in an analytics job [124]. Since serverless computing tasks are short-lived and any data that a task stores locally is lost when the task terminates, serverless analytics jobs share data using distributed storage solutions. However, popular storage systems used to share data in serverless computing applications today (e.g., Amazon S3 [22] and Redis [128]) either introduce significant performance overhead or they require users to manually manage expensive storage clusters.

Pocket is a storage service for ephemeral data sharing, designed specifically for serverless analytics and intended to be operating by cloud providers. More generally, Pocket is a distributed temporary data store that a variety of cloud applications can use for fast, elastic ephemeral storage. Pocket's design spans three different planes, which can each be scaled independently: i) a data plane that stores data objects across multiple nodes and storage technologies, ii) a metadata plane that routes client read/write requests to the right storage nodes, and iii) a control plane that decides which resources to allocate for each job and how to autoscale the storage cluster based on resource utilization. The system leverages optional hints, which users of application frameworks can provide through Pocket's client API, to determine a job's latency, throughput and capacity requirements. When a job registers with the Pocket controller, Pocket uses heuristics to translate any provided hints about the application's characteristics to a cost-effective storage resource allocation that satisfies the job's performance requirements. Storage nodes consist of a variety of technologies (e.g., DRAM, Flash, and disk) with different performance-cost trade-offs and run optimized software for network storage processing. For example, Pocket leverages ReFlex for fast access to Flash storage from Pocket clients.

Pocket reduces the average time that serverless tasks in a video analytics application spend on ephemeral I/O by over 4× compared to Amazon S3, a popular object storage service used by many serverless cloud applications. Pocket offers applications similar performance to Redis, a popular memory-based data store, while offering elastic resource provisioning and saving close to 60% in cost. These cost savings come from intelligent data placement across storage technologies and Pocket's pay-what-you-use cost model as a cloud provider managed storage service.

Selecta: Automatic Storage Resource Allocation

While Pocket rightsizes resources in response to load and relieves users from managing storage resources, the system still relies on users or application frameworks to provide hints about application characteristics to achieve cost-effective resource allocations. The third system we present automates storage and compute resource allocation decisions for data analytics jobs without relying on user hints.

Our system, Selecta, recommends a near-optimal resource allocation for a target application by predicting the application's execution time across a variety of cloud resource configurations. The system learns by leveraging performance data collected for applications that were previously run across a variety of compute and storage configurations. Using a technique called latent factor collaborative filtering, Selecta makes fast, accurate recommendations with sparse training data. Collaborative filtering works by uncovering latent similarities across jobs and resource configurations. The advantage of collaborative filtering is that users do not need to engineer features to train an accurate model and the approach is known to work well with sparse input data.

Selecta can find resource allocations that minimize the execution time of a job, minimize the cost of running a job, or maximize the performance of a job within a budget. For minimizing job execution time, Selecta has a 94% probability of recommending a configuration that performs within 10% of the true highest performance configuration. For minimizing job execution cost, where cost is a function of resource pricing per unit time and the job execution time, Selecta has an 80% probability of recommending a configuration that achieves within 10% of the true optimal cost configuration. Though we present Selecta as a tool for cloud users who need to decide between a plethora of virtual machine and storage configurations available in the cloud, the tool can also be used by cloud providers to provision resources for services like serverless computing in which the provider is responsible for resource allocation. As data-intensive workloads continue to grow in complexity and cloud options for compute and storage increase, automating storage resource management will become increasingly useful for end users and cloud providers.

1.2 Thesis Organization

The rest of this thesis is organized as follows. Chapter 2 provides relevant background and motivation. Chapter 3 describes ReFlex, a software system for fast, predictable remote access to Flash storage over commodity cloud networks. In Chapter 4, we describe Pocket, a system that builds on top of ReFlex's fast dataplane to offer a fast, elastic ephemeral data store for an emerging class of cloud applications known as serverless analytics. In Chapter 5, we present Selecta, a system that automates compute and storage resource allocation by recommending near-optimal cloud resource configurations based on sparse performance data collected across jobs. We conclude with a summary of our main contributions and a discussion of future work directions in Chapter 6.

Chapter 2

Background and Motivation

2.1 Cloud Storage

Much of today's economy relies on data, as companies capture, process, and capitalize on data in every step of their supply chain. The volume of data that organizations are generating and analyzing is growing exponentially. A recent study estimates that the amount of data being recorded worldwide is increasing 30% to 40% per year [129]. A significant portion of the data stored is also being actively processed. While consumers are using online services such as social media, video streaming, and cloud storage on a regular basis, businesses are mining vast amounts of customer data to provide greater levels of personalization. At this rate, it is predicted that in the year 2025, we will process five zettabytes of data on a global scale [169].

To store and process growing volumes of data, organizations and individuals are increasingly turning to the cloud. Cloud computing offers four key benefits for storing and processing data: high performance, resource elasticity, cost efficiency, and high availability. Cloud users can scale their resource usage up or down as the compute and storage demands of applications vary over time and pay only for the resources consumed at each time interval. The cost of operating highly available compute and storage infrastructure is amortized as cloud operators share their large-scale platforms among multiple tenants.

However, users only reap performance and cost efficiency benefits from the cloud if the cloud platform can provide each application with the right type and amount of compute and storage resources. In particular, the choice of storage is critical for the performance and cost of data-intensive cloud applications. For example, a Spark SQL equijoin query on two 128 GB tables completes 8.7× faster and costs 8× less when using local NVMe Flash storage compared to remote disk volumes on the Amazon EC2 public cloud [161, 123].

To address the diverse storage requirements of applications, today's cloud providers offer a wide variety of storage options. Object storage, distributed file storage, and block storage are common

classes of storage systems. Object storage (e.g., Amazon S3 [22]) enables access to data via a REST API for ease of use and access control. The flat namespace of object stores also makes them highly scalable. Distributed file storage systems (e.g., Azure Files [144]) are designed for sharing data in a hierarchical namespace between multiple tenants. Block storage (e.g., Google Compute Engine Persistent Disks [85]) provides raw access to storage devices for optimal performance and is the most common storage interface used by database systems. We can further classify storage systems based on the underlying hardware they use to store data, which can consist of hard disks, SATA/SAS solid-state drives, high performance NVMe Flash devices, or more recently non-volatile memory. These devices may be local to the cloud instances running the applications or remote. These options alone lead to storage configurations that can differ by orders of magnitude in terms of throughput, latency, and cost per bit.

Hardware Trends in Storage and Networking

The cloud storage landscape is heavily influenced by advances in datacenter hardware. In particular, we consider the increasing popularity of Flash storage devices and the vast improvements in networking hardware throughout and latency.

NAND Flash storage devices are replacing disks for an increasing number of applications in the cloud. Transistor scaling, multi-level cell technology and 3D integration have delivered a continuous increase in capacity, while new Flash controllers have leveraged high degrees of architectural parallelism and new software interfaces such as Non-Volatile Memory Express (NVMe) to significantly increase performance [41, 154]. As a result, Flash devices now provide up to one million I/O operations per second (IOPS) and read latencies as low as 20 μs [175]. Flash storage is also more power efficient than traditional rotating disk storage and its lack of moving parts improves reliability.

In addition to advances in NAND storage technology, improvements in datacenter networks are stimulating new use cases for Flash storage in the cloud. Public cloud platforms have transitioned from 1 Gigabit Ethernet to 10 Gigabit Ethernet interconnect in the past decade and are now undertaking a similar transition towards 100 Gigabit Ethernet. Such improvements to networking infrastructure are enabling low latency, high throughput access to remote Flash for large-scale distributed storage. For example, at Facebook, many applications that generate web-page content use NVMe Flash to host large-scale persistent key-value storage services. Similarly, LinkedIn reports using NVMe SSDs to scale its distributed key-value database, Project Voldemort [136], to process over 120 billion relationships per day [102]. NVMe Flash is commonly deployed in datacenter servers as direct-attached storage on the PCIe bus.

While the performance of Flash storage and datacenter networking hardware has dramatically improved over the past decade, the software stacks for storage and network processing in contemporary operating systems have failed to keep pace with the hardware. Operating systems running on today's cloud servers were designed decades ago with the assumption that I/O is slow [38]. This

assumption was correct when mechanical disk seeks and network propagation delays took orders of magnitude longer than system calls, cache miss handling, and interrupt request delivery [36]. However, as we describe in Section 2.3, the recent improvements in storage and networking hardware require us to reconsider the design of operating system software to satisfy the performance requirements of today's cloud applications. We begin by analyzing and summarizing the storage requirements of current and emerging cloud applications.

2.2 Understanding Cloud Application Storage Requirements

Cloud applications such as search, social networking, and big data analytics are redefining the requirements for operating systems software and storage systems. A single cloud application can consist of hundreds of software services, deployed on thousands of servers. At a high level, the new requirements for storage systems include microsecond-level responses to remote requests with tight tail latency guarantees, high throughput to support millions of concurrent requests, and quality of service for multiple tenants sharing storage resources [65, 31]. Elastic resource provisioning is also important to adapt to dynamic variations in application resource requirements.

Cloud applications vary widely with respect to their compute and storage resource requirements. We analyze three important classes of cloud applications to understand their needs and the implications for storage system design. First, we study large-scale, multi-tier web applications that rely on distributed key-value stores to serve requests. Next, we characterize the heterogeneous data streams in big data analytics applications. Finally, we outline the storage properties required to enable an emerging class of applications, known as serverless analytics, to exchange ephemeral data efficiently in the cloud.

2.2.1 Key-Value Storage for Multi-Tier Web Applications

Large-scale web applications, such as social networking and e-commerce services, are commonly deployed with a three-tier architecture consisting of a web, application, and datastore tier. The web tier is responsible for serving web page content to users. Personalized web content is generated by the application tier, which reads, aggregates, and processes information from the datastore tier. For our analysis of web application storage requirements, we focus on the datastore tier as this tier hosts terabytes and even petabyte-scale persistent key-value store services. These key-value stores are used to maintain a variety of user state such as viewing history. Examples of popular persistent key-value stores embedded in cloud applications include RocksDB, LevelDB, and Project Voldemort [83, 75, 136].

Table 2.1 summarizes the I/O characteristics of four real datastore tier workloads at Facebook, as observed from 2014 to 2015, which use a Flash storage based RocksDB key-value store. These applications typically issue 2,000 to 10,000 read operations to Flash storage per second per TB.

Table 2.1: Facebook Flash workload characteristics

Workload	Read IOPS/TB	Avg. read size (kB)	Write IOPS/TB	Avg. write size (KB)
A	2k - 10k	10	100	500
B	2k - 6k	50	1000	700
C	2k - 4k*	15	500	2000
D	2k - 4k	25	150	500

*Spikes of 75K read IOPS are also observed 1% of the time

Reads are random access and generally under 50 KB in size. Write operations occur when RocksDB flushes an in-memory data structure or performs a database compaction. Writes are sequential and significantly larger in size (up to 2 MB) but less frequent than reads. Datastore servers hosting these applications with direct-attached NVMe Flash saturate CPU cores before saturating Flash IOPS due to the computational intensity of datastore services. The CPU cycles consumed handling complex queries, compactions, and serializing or deserializing data limit I/O rates on datastore servers to only a fraction of the capabilities of today's NVMe Flash devices. Despite these workloads underutilizing NVMe Flash devices, their low latency requirements and ratio of storage throughput to capacity still makes NVMe Flash the more suitable storage technology compared to disk for this important class of cloud applications.

Figure 2.1 shows the Flash storage capacity, Flash read throughput, and CPU utilization over time in a production cluster hosting a real Facebook application (Workload A in Table 2.1) using the RocksDB key-value store. Each metric is normalized to its maximum value over the six month period. The plot shows that Flash capacity, Flash throughput, and CPU utilization vary dramatically over time and follow separate trends. For example, in mid-March, a change to the application code dramatically improved CPU efficiency and hence decreased CPU utilization, however storage capacity and throughput continued to increase due to higher application load. Resource utilization studies of the Microsoft Azure cloud computing platform and a large private cloud at Google have also shown that storage capacity, I/O rates, and CPU usage are not necessarily correlated [137, 194].

The unique and often dynamically varying compute and storage requirements of applications makes designing server machines with the right balance of CPU and Flash storage difficult [121, 35]. The lack of balance leads to deploying machines with significantly overprovisioned resources, which can increase the total cost of ownership [97]. For example, in Figure 2.1, Flash capacity and throughput are under-utilized for long periods of time. Overprovisioning IOPS is particularly common when deploying NVMe Flash in datacenters because software overheads for inter-tier communication and query processing in storage services often saturate CPU cores before saturating NVMe Flash IOPS on modern devices [117]. Baidu reported their storage system's realized I/O bandwidth was only up to 50% of their Flash hardware's raw bandwidth [157]. Similarly, Flash capacity is often underutilized as servers are deployed with enough Flash to satisfy projected future demands and amortize the cost of the Flash controller on each device.

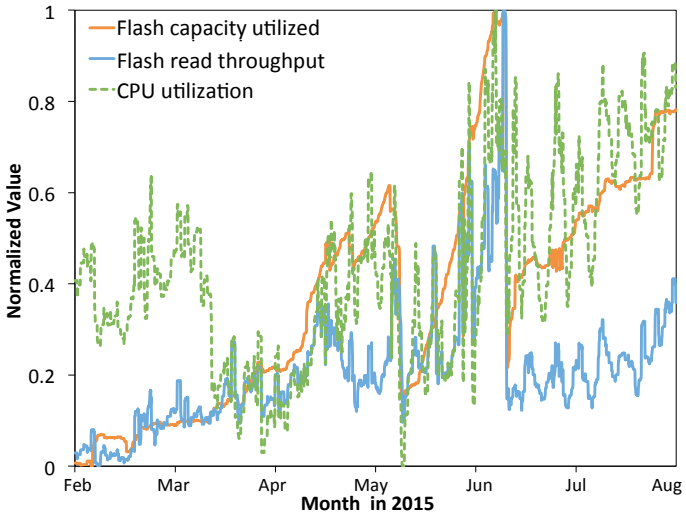


Figure 2.1: Sample resource utilization on servers hosting a Flash-based key-value store service at Facebook, normalized over a 6 month period. Flash and CPU utilization vary over time and scale according to separate trends.

Implications for Storage System Design: *Flash Storage Disaggregation*

Resource disaggregation is a known way to address the challenge of imbalanced resource requirements and the resulting overprovisioning of datacenter resources [98]. By physically decoupling compute and Flash storage resources, datacenter operators can more easily customize their infrastructure to maximize the performance-per-dollar for target workloads. We use the term "disaggregated Flash" or "remote Flash" to refer to Flash that is accessed over a high-bandwidth network, as opposed to Flash accessed locally over a PCIe link. Remote Flash accesses can be served by a high-capacity Flash array on a machine dedicated to serving storage requests over the network. Alternatively, we can enable remote access to Flash on a nearby server, which itself runs Flash-based applications, but has spare Flash capacity and IOPS that can be shared over the network. With either approach, disaggregation can span racks, rows, or the whole datacenter.

Section 2.3.1 outlines the challenges associated with disaggregating modern Flash storage devices from compute resources. In Chapter 3, we present a software-based system for remote Flash access that provides low latency, high throughput, and quality of service with low CPU requirements.

2.2.2 Heterogeneous Data Streams in Big Data Analytics

Data analytics are another important class of data-intensive workloads commonly run in the cloud. In this section, we focus on large-scale data analytics applications deployed on popular cluster computing frameworks, such as Spark, Hadoop, and Storm [217, 210, 1]. In Section 2.2.3, we will

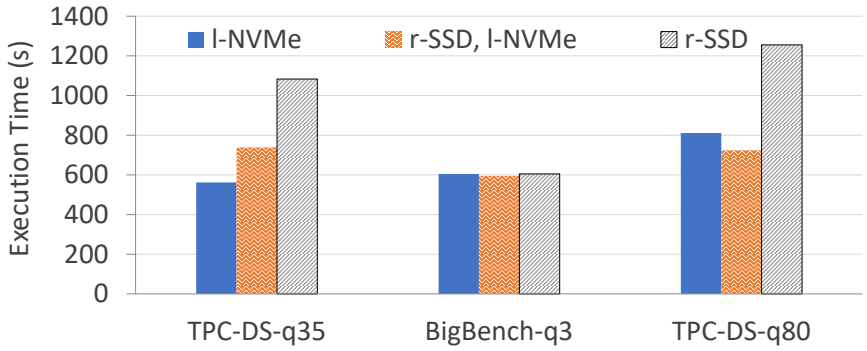


Figure 2.2: Performance of three applications on an 8-node cluster with different storage configurations. TPC-DS-q35 is I/O-bound, performing best with local NVMe Flash storage (l-NVMe). BigBench-q3 is CPU-bound and performs equally well with all storage options, including remote SSD (r-SSD). TPC-DS-q80 performs best with the hybrid storage option (r-SSD, l-NVMe), which stores intermediate data on local NVMe and input/output data on remote SSD to minimize interference.

discuss data analytics applications deployed on serverless cloud computing platforms, which are becoming increasingly popular. In both types of deployments, it is common for analytics jobs to be run periodically as new data becomes available. For example, studies at Microsoft have shown that 40% of jobs in production data analytics clusters are recurring, as queries are executed repeatedly on newly arriving data [12, 76].

Distributed data analytics applications typically access heterogeneous data streams such as original input data for a job, final output data, intermediate data exchanged between tasks in a job, and various logs. These data streams have distinct characteristics in terms of access frequency, access patterns, and data lifetime. Hence, each data stream may be better served by different types of storage devices. For example, input/output data is generally long-lived and sequentially accessed, whereas intermediate data is short-lived and most accesses are random. Thus input/output files for data analytics jobs are traditionally stored in a distributed filesystem (e.g., HDFS [187]) or object storage systems (e.g., Amazon S3 [22]). Intermediate data is typically read/written to/from a dedicated local Flash storage volume on each node, which is managed by a long-running cluster computing framework agent. Intermediate data is spilled to remote disk if extra capacity is needed.

Determining the right cloud storage configuration for performance and cost efficiency is difficult, particularly when considering the needs of heterogeneous data streams in a distributed data analytics application and the many storage options available in the cloud. As an example, Figure 2.2 compares the performance of three Spark applications using eight `i3.xl` AWS instances with local NVMe Flash storage (l-NVMe), remote Flash storage (r-SSD), and a hybrid storage configuration that uses r-SSD for input/output data and l-NVMe for intermediate data. The first application is I/O-bound and benefits from the high throughput of NVMe Flash. The second application has a CPU bottleneck and thus performs the same with all three storage options. The third application is I/O-bound

and performs best with the hybrid storage option since it minimizes interference between read and write I/Os, which have asymmetric performance on Flash [122]. While this result is not surprising after careful consideration of the characteristics of input/output and intermediate data, selecting the optimal storage configuration is far from trivial.

In addition to the storage configuration, cloud users must choose from a variety of virtual machine types in the cloud. This involves deciding the right number of CPU cores and memory, the number of instances, and their network bandwidth. These choices often affect storage and must be considered together. For example, network limits can constrain throughput when accessing remote storage volumes. As new generations of processors and networking technology are deployed in datacenters along with emerging storage technology such as non-volatile memory, the cloud configuration landscape is only becoming more diverse and complex.

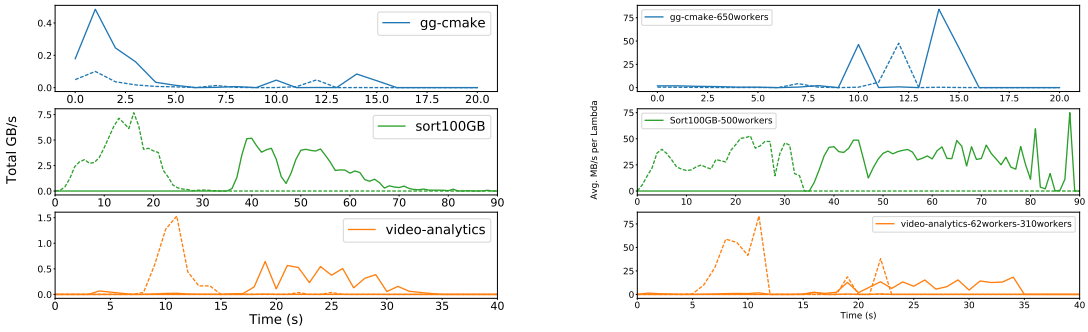
Implication for Storage System Design: *Automate Storage Resource Allocation*

As more and more users migrate their applications to the cloud and the compute and storage configuration space continues to expand with new deployments of networking and storage technologies, selecting the right configuration for each individual application is becoming increasingly challenging. It is thus becoming increasingly desirable to automate storage resource allocation decisions.

The fact that users commonly run analytics jobs repeatedly in the cloud as new data arrives [137] opens opportunities to build predictive models of application performance across configurations based on prior data. For example, information about how a job performed on a few particular resource configurations can help predict how the job will perform on other configurations, so that we can make the optimal choice of configuration to maximize performance and/or cost objectives. One of the major challenges is that we have sparse training data, as the goal is to minimize the number of configurations across which an application needs to be profiled in the large configuration space. In Chapter 5, we present a tool that recommends near-optimal configurations of compute and storage resources in the cloud for data analytics workloads based on sparse data collected by profiling training workloads.

2.2.3 Ephemeral Storage for Serverless Analytics

A new execution model, called serverless computing, is becoming increasingly popular in the cloud. With services such as AWS Lambda, Google Cloud Functions, and Azure Functions, cloud users can now write applications as collections of stateless functions which they deploy directly to a serverless framework instead of running tasks on traditional virtual machines with pre-allocated resources [23, 87, 145, 5]. In this execution model, the cloud provider schedules user tasks onto physical resources with the promise of automatically scaling resources according to application demands and charging users only for the fine-grain resources their tasks consume. These frameworks conveniently abstract away the complexities of server infrastructure management for cloud users.



(a) Total job throughput over time.

(b) Average throughput per lambda over time.

Figure 2.3: Ephemeral storage throughput requirements vary dynamically over time for serverless analytics applications. Solid lines show read throughput while dotted lines plot write throughput.

While already popular for web microservices and IoT applications, the elasticity and fine-grain billing advantages of serverless computing are also appealing for a broader range of applications, including interactive data analytics. Several frameworks are being developed which leverage serverless computing to exploit high degrees of parallelism in analytics workloads and achieve near real-time performance [78, 116, 64].

We study three different serverless analytics applications and characterize their storage throughput and capacity requirements [124]. We use AWS Lambda as our serverless platform and configure serverless tasks with the maximum supported memory (3 GB) [23]. We refer to serverless tasks as "lambdas". Figure 2.3a plots each job's total storage bandwidth usage over time, summed over all lambdas, while Figure 2.3b shows the average per-lambda throughput over time. The solid lines show the read throughput while the dotted lines plot write throughput. The plots show that application storage throughput demands vary dynamically over time, requiring high resource elasticity to achieve high performance while minimizing cost. We describe each job below.

Parallel software build: We use a framework called `gg` to automatically synthesize the dependency tree of a software build system and coordinate lambda invocations for distributed compilation [77, 6]. Each lambda fetches its dependencies from ephemeral storage, computes (i.e., compiles, archives or links depending on the stage), and writes an output file. Compilation stage lambdas read source files which are generally up to 10s of KBs. While 55% of files are read only once (by a single lambda), others are read hundreds of times (by many lambdas in parallel), such as glibc library files. Lambdas which archive or link read objects up to 10s of MBs in size. We use `gg` to compile `cmake` which has 850 MB of ephemeral data.

MapReduce Sort: We implement a MapReduce style sort application on AWS Lambda, similar to PyWren [116]. Map lambdas fetch input files from long-term storage (S3) and write intermediate files to ephemeral storage. Reduce lambdas merge and sort intermediate data read from ephemeral

storage and write output files to S3. Sorting is I/O-intensive. For example, we measure up to 7.5 cumulative GB/s when sorting 100 GB with 500 lambdas. Each intermediate file is written and read only once and its size is directly proportional to the dataset size and inversely related to the number of workers.

Video analytics: We use Thousand Island Scanner (THIS) to run distributed video processing on lambdas [167]. The input is an encoded video that is divided into batches and uploaded to ephemeral storage. First stage lambdas read a batch of encoded video frames from ephemeral storage and write back decoded video frames. Each lambda then launches a second stage lambda which reads a set of decoded frames from ephemeral storage, computes a MXNET deep learning classification algorithm and outputs a classification result. We use a video consisting of 6.2K 1080p frames and tune the batch size to optimize runtime (62 lambdas in the decode stage and 310 lambdas for classification). The total ephemeral storage capacity is 6 GB.

Implications for Storage System Design: *Elastic Ephemeral Storage*

The newfound elasticity and fine grain resource usage billing that serverless platforms provide for computation motivates equally elastic storage solutions, particularly for intermediate data sharing between tasks. We outline several storage requirements for the design of distributed ephemeral storage based on our analysis of serverless analytics applications. As applications can consist of thousands of lambdas in one execution stage and only a few lambdas in another, the storage system should have *high elasticity* to adjust to application load. Since the granularity of data access varies widely, storing both small and large objects should be cost and performance efficient. *Low latency* and *high IOPS* are important for accesses to small objects, while *high throughput* is critical for accesses to large objects. To relieve users from the difficulty of managing storage clusters, the storage service should *autoscale resources* based on load and charge users based on the bandwidth and capacity their applications consume. This extends the serverless abstraction to storage. Finally, the storage system can leverage the unique characteristics of ephemeral data. Namely, ephemeral data is short-lived and can easily be re-generated by re-running a job's tasks. Thus, unlike traditional long-term storage, an ephemeral storage system can provide low data durability guarantees. Furthermore, since the majority of ephemeral data is written and read only once (e.g., a mapper writes intermediate results for a particular reducer), the storage system can optimize capacity usage with an API that allows users to hint when data should be deleted right after it is read.

In Chapter 4, we present Pocket, an elastic distributed data store that allocates and autoscales storage cluster resources to enable fast, elastic ephemeral data exchange between serverless tasks.

2.3 Challenges for High Performance, Resource-Efficient Cloud Storage

To satisfy the requirements of applications, cloud storage systems need to balance providing high, predictable performance with minimizing cost. While achieving high performance with infinite resources is straightforward, optimizing for cost efficiency requires provisioning *just enough* resources to each application to satisfy its performance requirements. As outlined in Section 2.2, for the cloud applications we study, high performance entails both high throughput, to handle many concurrent requests, and low tail latency, to optimize per-request response times. Since the cost of buying server equipment dominates the total cost of ownership in datacenters, a promising way to reduce the cost of cloud storage is to optimize resource efficiency [97]. We refer to a resource-efficient cloud storage system as a storage system that is able to satisfy the storage capacity, throughput, and latency requirements of applications with as few resources as possible. Upon optimizing the resource allocations of individual applications, we can further improve cluster resource efficiency by bin-packing applications on shared cloud infrastructure. This amortizes resource costs and enables serving a greater number of users from a single cloud facility. Cloud storage must still provide quality of service when shared across multiple tenants.

Next, we summarize the challenges associated with the two major requirements for high performance, resource-efficient cloud storage: fast, predictable access to remote data and intelligent control of storage resource allocation. We describe why existing cloud storage solutions fall short.

2.3.1 Fast, Predictable Access to Remote Data

Remote access to hard disks is already common in datacenters [34, 84], since their high latency and low throughput easily hide network overheads [35, 25]. A variety of software systems can make remote disks available as block devices (e.g., iSCSI [178]), network file systems (e.g., NFS [176]), distributed file systems (e.g., Google File System [84]), or distributed data stores (e.g., Bigtable [50]). There are also proposals for hardware-accelerated, remote access to Flash using Remote Direct Memory Access (RDMA) (e.g., NVMe over Fabrics [52]) or PCIe interconnects [32, 201, 107].

Unfortunately, existing approaches face two main challenges for efficient access to remote Flash. First, network and storage request processing in traditional operating system software is too slow compared to the low latency and high throughput of modern Flash storage devices. Second, existing solutions do not manage read/write interference between multiple tenants sharing a Flash storage device, leading to unpredictable performance due to the asymmetry of read and write performance on Flash storage.

Existing software is too slow: Conventional remote server access over 10 Gigabit Ethernet and the TCP/IP network protocol adds at least 50 μs round-trip latency for a request before any software-based storage protocol processing even begins [38]. This latency overhead is significant as a local

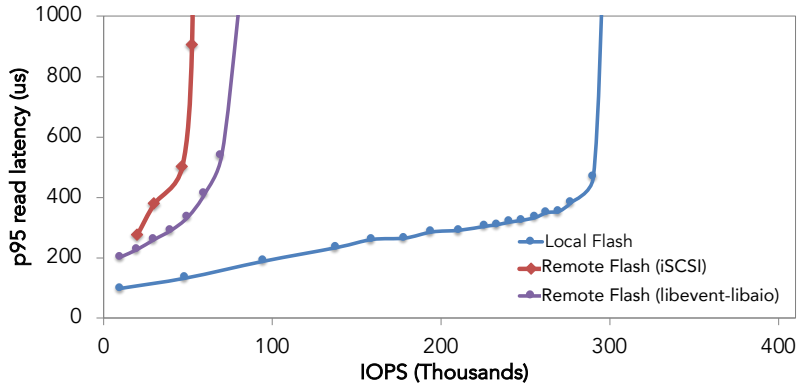


Figure 2.4: Remote versus local access to Flash storage with traditional systems software.

access to NVMe Flash takes 20-100 μs [56]. Block I/O protocols such as iSCSI, commonly used in storage attached network (SAN) architectures, introduce additional latency for protocol processing and copying data between kernel and user buffers [121, 115]. Network and storage protocol processing consumes significant compute resources to saturate the high throughput of an NVMe Flash storage device. For example, Figure 2.4 compares local versus remote access to Flash, using a single CPU core on the Flash storage server. Remote access with iSCSI or traditional Linux libraries `libevent` and `libaio` results in over 4× lower throughput and 2× higher latency than local access to Flash storage due to network/storage processing saturating the CPU cores on the remote Flash storage server. Higher throughput with iSCSI and `libevent-libaio` is attainable by using more CPU cores on the remote storage server, however these extra compute resources increase the cost of disaggregating Flash storage. Finally, using distributed file systems such as GFS and HDFS to access remote storage has high latency for kilobyte-sized requests as these systems are optimized for multi-megabyte data transfers on remote disk [84, 187]. On the other hand, network file sharing protocols used in network attached storage (NAS) architectures have limited scalability.

Read/write interference: Remote Flash is useful if storage devices can be shared between multiple tenants while still providing performance guarantees. Achieving predictable performance on shared NVMe Flash devices is difficult because Flash read performance is sensitive to interference from concurrent write requests. Due to the physical properties of NAND Flash storage technology, write operations take an order of magnitude longer than read operations to complete and they are typically not preemptible [43]. Requests attempting to read any page that is on a Flash chip service a write operation are blocked until the write completes, resulting in high tail latency for read operations. While applications using local Flash can be designed to cope with interference, this becomes a challenge for remote Flash systems in which multiple tenants sharing the same device are unaware of each other. Unfortunately, interference management is not addressed by hardware-accelerated schemes like NVMe over Fabrics [52], QuickSAN [48] or iSCSI extensions for RDMA (iSER) [49].

The current isolation features of NVMe devices, namely hardware queues and namespaces, are not sufficient to manage interference between multiple remote clients without additional software.

In Chapter 3, we introduce ReFlex, a system that enables disaggregating Flash storage to achieve resource efficiency benefits without sacrificing application performance compared to accessing local Flash storage.

2.3.2 Intelligent Control of Storage Resource Allocation

While a fast data plane for low latency, high throughput access to remote data provides the flexibility to customize resource allocations to application requirements, we also need an intelligent control plane to make these resource allocation decisions. There are several challenges for the design and implementation of such a control plane. First, making resource-efficient resource allocation decisions requires knowledge of application resource requirements, which can be difficult to determine in the first place [68, 195]. Even if the requirements are known, traditional storage interfaces make it difficult to pass this information to the storage layer [198, 11]. Second, as discussed in Section 2.2, application requirements vary dynamically. To maximize resource efficiency, provisioned resources must be autoscaled to accommodate these variations. Resource allocation decisions also need to be made quickly at a large scale, with potentially millions of decisions per second [69]. It is thus necessary to automate storage resource allocation and elastic scaling decisions.

Learning application storage requirements: The first step in automating storage resource management is learning application requirements. A storage system can learn application requirements by interpreting hints provided by users or application frameworks. Alternatively, the system can collect performance data across application runs and build a model to predict how an application will perform across various storage configurations. Both approaches have their challenges. For the former, accepting hints from users or application frameworks requires designing a storage API that is expressive enough to provide useful information to the system while being easy for users to use. Prior work has shown that users are not good at estimating the resource requirements of their applications and will often ask for more resources than required, resulting in over-provisioning [68]. Instead of explicitly specifying resource requirements for applications, it is more intuitive for users to specify high-level performance requirements. For resource-efficient storage allocation, it is particularly important for the system to learn the application's capacity, latency, and throughput requirements. We can relieve users from providing hints about their workloads by designing the storage control plane to learn from past performance data. The challenge then becomes getting accurate predictions from a model trained with very sparse input data. The training data is sparse because we would like our system to select the (near) optimal resource allocation after profiling each job on as few configurations as possible. In Chapter 5, we address this challenge by applying collaborative filtering, a machine learning approach known to work well on sparse training sets.

Resource elasticity: After the storage system learns application characteristics and translates

these into a resource allocation, the system still needs to continue monitoring application behaviour and respond with high resource elasticity. We define elasticity as the degree to which a system is able to adapt to workload changes by scale resources up and down in an automatic manner, such that at each point in time the available resources match the current demand as closely as possible [99]. Various reactive [81], predictive [57, 127, 171, 72, 152, 202, 208] and hybrid [53, 182, 103, 80, 150] approaches have been proposed to automatically scale resources based on demand at both the cluster and single-node level [168, 151]. Reactive approaches are difficult to implement correctly due to the delay between observing changes in application behaviour and taking action. Predictive approaches, on the other hand, typically require large and representative datasets to train accurate models. Much prior work focuses on one to two resource dimensions (e.g., CPU cores and memory) to make autoscaling decisions. However, resource allocation and/or scaling is a high-dimensional problem involving compute, memory, storage, and network resources. While some systems take the approach of rightsizing applications at the granularity of virtual machines [100], finer grain resource elasticity can further improve resource efficiency.

Data placement: Once we have intelligent, dynamic allocation of resources, the question of where to store what data remains. Data placement should take into account heterogeneous data streams in applications, as discussed in Section 2.2.2, and hardware capabilities, particularly if the storage allocation consists of heterogeneous resources such as DRAM, Flash, and disk which each offer different performance-cost trade-offs. Common data placement approaches based on hashing object names, such as consistent hashing [118], are not effective for ephemeral data in elastic storage clusters. Consistent hashing still requires some data to be migrated and redistributed across nodes when the cluster size changes [193]. For short-lived data (e.g., data that lives for less than a minute), migration has high overhead and should be avoided. Existing storage control planes that manage data placement, such as Mirador [211] and Extent-Based Dynamic Tiering [90], focus on long-term as opposed to ephemeral data storage. In cluster computing, ephemeral data is canonically stored on local storage devices rather than distributed storage services, due to the low latency and high throughput required when accessing ephemeral data in applications. However, with advances in networking technology and software stacks for optimized network/storage processing, we can now design distributed storage services that are fast enough to satisfy the performance requirements of ephemeral data and have the benefit of being elastically scalable across nodes. In Chapter 4, we present Pocket, a fast and elastic ephemeral data store that places data across multiple storage tiers across cluster nodes to enable efficient data shuffling for serverless analytics applications.

Chapter 3

ReFlex: Remote Flash ≈ Local Flash

3.1 Introduction

NVMe Flash devices deliver up to 1 million I/O operations per second (IOPS) at sub 100μs latencies, making them the preferred storage medium for many data-intensive, online services. However, as discussed in Section 2.2, the Flash devices deployed in datacenters are often underutilized in terms of capacity and throughput due to the imbalanced requirements across applications and over time. In general, it is difficult to design machines with the perfect balance between CPU, memory, and Flash resources for all workloads, which leads to over-provisioning and higher total cost of ownership (TCO). Remote access to Flash over the network can greatly improve utilization by allowing access to Flash on any machine that has spare capacity and bandwidth.

Implementing remote access to Flash is challenging for the reasons we discussed in Section 2.3.1. First, existing operating system software is too slow in processing remote storage requests compared to the latency and throughput capabilities of modern network and storage hardware. Second, the interference of read and write requests between multiple tenants sharing a Flash device leads to unpredictable performance. Finally, it is useful to have flexibility in the degree of sharing, the deployment scale, and the network protocol used for remote connections. Recently proposed, hardware-accelerated options, such as NVMe over RDMA fabrics [52], lack performance isolation and provide limited deployment flexibility.

We present ReFlex[1], a software-based Flash storage server that implements remote Flash access at a performance comparable with local Flash accesses. ReFlex achieves high performance with limited compute requirements using a novel dataplane kernel that tightly integrates networking

[1]ReFlex is open-source software, available at: `https://github.com/stanford-mast/reflex`.

and storage. The dataplane design avoids the overhead of interrupts and data copying, optimizes for locality, and strikes a balance between high throughput (IOPS) and low tail latency. ReFlex includes a QoS scheduler that implements priorities and rate limiting in order to enforce service level objectives (SLOs) for latency and throughput for multiple tenants sharing a device. ReFlex provides both a user-level library and a remote block device driver to support client applications.

The ReFlex server achieves 850K IOPS per core over commodity 10GbE networking with TCP/IP. Hence, it can serve several NVMe devices and meet networking line rates at low cost. Its unloaded latency is only 21μs higher than direct access to local Flash through NVMe queues. The ReFlex server can support thousands of remote tenants. Its QoS scheduler can enforce the tail latency and throughput requirements of tenants with SLOs, while allowing best-effort tenants to consume all remaining throughput of the NVMe device. Finally, using legacy applications, we show that even with heavy-weight clients, ReFlex allows for performance levels nearly identical to those with local Flash.

3.2 ReFlex Design

ReFlex provides low latency and high throughput access to remote Flash using a dataplane architecture that tightly integrates the networking and storage layers. It serves remote read/write requests for logical blocks of any size over general networking protocols like TCP and UDP. While predominately a software system, ReFlex leverages hardware virtualization capabilities in NICs and NVMe Flash devices to operate directly on hardware queues and efficiently forward requests and data between NICs and Flash devices without copying. Its polling-based execution model (§3.2.1) allows requests to be processed without interruptions, improving locality and reducing unpredictability. ReFlex uses a novel I/O scheduler (§3.2.2) to guarantee latency and throughput SLOs for tenants with varying ratios of read/write requests. ReFlex can serve thousands of tenants and network connections, using as many cores as needed to saturate Flash device IOPS.

3.2.1 Dataplane Execution Model

Each ReFlex server thread uses a dedicated core with direct and exclusive access to a network queue pair for packet reception/transmission and an NVMe queue pair for Flash command submission/-completion.

Figure 3.1 reviews the execution model for a ReFlex server thread processing an incoming Flash read (or write) request. First, the NIC receives a network packet and delivers it via DMA to a pre-allocated memory buffer provided by the software networking stack ①. The ReFlex thread polls the receive descriptor ring and processes the packet through the Ethernet driver and networking stack (e.g., TCP/IP), generating event conditions indicating the availability of a new message ②. The same thread uses `libix` [38], a library similar to Linux `libevent` [165], to process the event. This

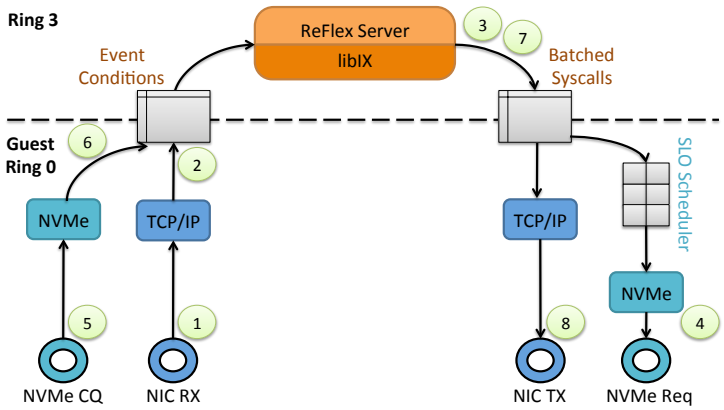


Figure 3.1: ReFlex per-thread execution model for processing requests on remote Flash server. Requests arrive on the NIC RX queue and replies are sent from the NIC TX queue after processing.

involves switching to the server code that parses the message, extracts the I/O request, performs access control checks and any other storage protocol processing required before submitting a Flash read (write) system call ③. The thread then switches to system call processing and performs I/O scheduling to enforce SLOs across all tenants sharing the ReFlex server (§3.2.2). Once scheduled, the request is submitted to the Flash device through an NVMe submission queue ④. The Flash device performs the read (write) I/O and delivers (retrieves) the data via DMA to (from) a pre-allocated user-space buffer ⑦. The thread polls the completion queue ⑤ and delivers a completion event ⑥. The event callback executes through `libix` and emits a send system call ⑦. Finally, the thread processes the send system call to deliver the requested data back to the originator through the network stack ⑧. The execution model supports multiple I/O requests per network message and large I/Os that span across multiple network messages.

This execution model achieves low latency for remote Flash requests. It runs to completion the two steps of request processing, the first between network packet reception and Flash command submission (① - ④) and the second between Flash completion and network packet transmission (⑤ - ⑧), without any additional interruptions or thread scheduling. Running to completion eliminates latency variability and improves data cache locality for request processing. ReFlex's two-step run to completion model avoids blocking on Flash requests. ReFlex avoids interrupt overhead by polling for network packet arrivals and Flash completions. Moreover, ReFlex implements zero-copy by passing pointers to the buffers used to DMA data from the NIC or Flash device.

In addition to the benefit from lower latency, ReFlex improves the throughput of remote Flash requests using two methods. First, it uses asynchronous I/O to overlap Flash device latency (50μs or more) with network processing for other requests. Once a command is submitted to the Flash device ④, the thread polls the NVMe completion queue for previously issued requests that require outgoing

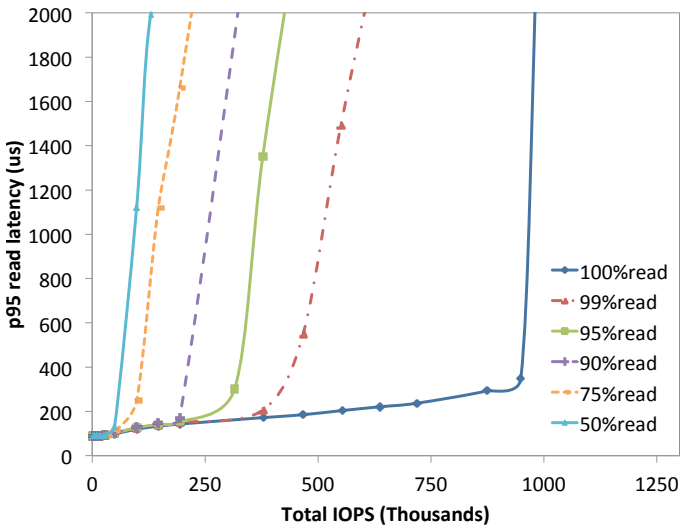


Figure 3.2: Read/write performance interference on Flash. Tail read latency depends on total IOPS and the read/write ratio.

network processing and polls the NIC receive queue for incoming packets that require incoming network processing. As long as there is work to do, the thread does not idle. Second, ReFlex employs adaptive batching of requests in order to amortize overheads and improve prefetching and instruction cache efficiency [38]. Under low load, incoming packets or completed NVMe commands are processed immediately without any delay. As load rises, the NIC receive and NVMe completion queues fill up and provide the opportunity to process multiple incoming packets or multiple completed accesses in a batch. The batch size increases with load but it is capped to 64 to avoid excessive latencies. Unlike conventional batching, which trades off latency for bandwidth, adaptive batching achieves a good balance between high throughput and low latency [38].

ReFlex scales to multiple threads, each using a dedicated core and separate hardware queue pairs. Threads need to coordinate only when issuing accesses to the NVMe command queue so that they collectively respect the tail latency and throughput SLOs of the various tenants that share the ReFlex server (see §3.2.2). A local control plane periodically monitors the load of all ReFlex threads and their ability to achieve the requested SLOs in order to increase or decrease the number of ReFlex threads. When the number of threads changes, remote tenants and network connections are rebalanced across threads as explained in [164]. Rebalancing takes a few milliseconds and does not lead to packet dropping or reordering.

3.2.2 QoS Scheduling and Isolation

The QoS scheduler allows ReFlex to provide performance guarantees for tenants sharing the Flash device(s) in a server. A *tenant* is a logical abstraction for accounting for and enforcing service-level

objectives (SLOs). An SLO specifies a tail read latency limit at a certain throughput and read-write ratio. For example, a tenant may register an SLO of 50K IOPS with 200μs read tail latency (95$^{\text{th}}$ percentile) at an 80% read ratio. In addition to such *latency-critical (LC) tenants*, which have guaranteed allocations in terms of tail latency and throughput, ReFlex also serves *best-effort (BE) tenants*, which can opportunistically use any unallocated or unused Flash bandwidth and tolerate higher latency. A tenant definition can be shared by thousands of network connections, originating from different client machines running any number of applications. An application can use multiple tenants to request separate SLOs for different data streams.

Figure 3.2 plots the tail read latency (95$^{\text{th}}$ percentile) on Flash as a function of throughput (IOPS) for workloads with various read-write ratios. As shown in Figure 3.2, enforcing an SLO on Flash device accesses is complicated by two factors. First, the maximum bandwidth (IOPS) the device can support depends on the overall read/write ratio of requests it sees across all tenants. Second, the tail latency for read requests depends on both the overall read/write ratio and the current bandwidth load. This behavior is typical for all NVMe Flash devices we have tested because write operations are slower and trigger activities for wear leveling and garbage collection that cannot always be hidden. Hence, the QoS scheduler requires global visibility and control over the total load on Flash and the type of outstanding I/O operations. We use a request cost model to account for each Flash I/O's impact on read tail latency (§3.2.2) and a novel scheduling algorithm that guarantees SLOs across all tenants and all dataplane threads (§3.2.2). ReFlex does not assume a priori knowledge of workloads.

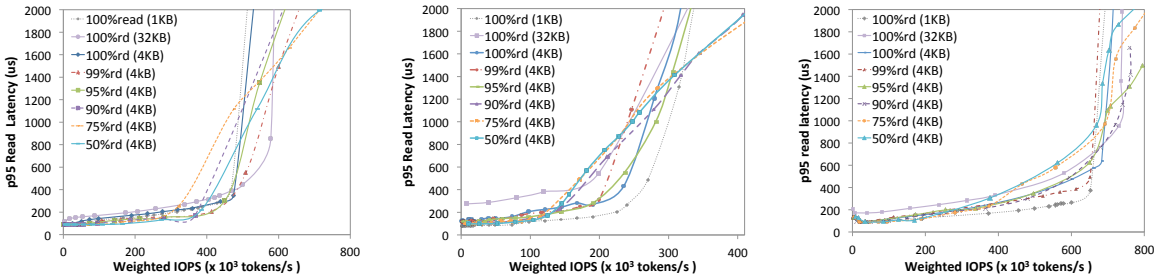
Request Cost Model

The model estimates read tail latency as a function of *weighted* IOPS, where the cost (weight) of a request depends on the I/O size, type (read vs. write), and the current read/write request ratio r on the device:

$$\text{I/O cost} = \left\lceil \frac{\text{I/O size}}{\text{4KB}} \right\rceil \times C(\text{I/O type}, r)$$

Cost is a function of r because some Flash devices provide substantially higher IOPS for read-only loads (r=100%) compared to 99% or lower read loads, as seen in Figure 3.2. Hence, the model adjusts the cost of read requests when the device load is read-only. Costs are expressed in multiples of tokens, where one token represents the cost of a 4KB random read request. In all Flash devices we have used, cost scales linearly with request size for sizes larger than 4KB (e.g., a 32KB request costs as much as 8 back-to-back 4KB requests). Cost is constant for requests 4KB and smaller, as these Flash devices seem to operate at 4KB granularity.

We calibrate the cost model for each type of Flash device deployed in the ReFlex server. First, we measure tail latency versus throughput with local Flash for workloads with various read/write ratios and request sizes (see 4KB example in Figure 3.2). Since the cost of write requests depends on the frequency of garbage collection and page erasure events, we conservatively use random write patterns



(a) Device A Request Cost Model (b) Device B Request Cost Model (c) Device C Request Cost Model

Figure 3.3: Request cost models for various workloads on 3 different NVMe Flash devices. We model read tail latency as a function of weighted IOPS, which is the total number of I/O operations per second with each I/O operation assigned a weight based on the type of the request (read vs. write), the I/O size, and the current read/write ratio on the device.

to trigger the worst case. Next, we use curve fitting to derive $C(\text{I/O type}, r)$. The model can be re-calibrated after deployment to account for performance degradation due to Flash wear-out [44].

Figure 3.3 shows latency versus weighted IOPS (tokens) plots for three different NVMe devices from multiple vendors and representing multiple generations and capacities of Flash devices. Device A is the one characterized in Figure 3.2. The value of $C(\text{write}, r < 100\%)$ is 10, 20, and 16 tokens for devices A, B, and C respectively. This means that write operations are 10 to 20 times more expensive than read operations, depending on the device. For device A, when the load from all tenants is read-only, the cost of a 4KB read request is half a token (i.e., $C(\text{read}, r = 100\%) = \frac{1}{2}$ token). For all three devices, our linear cost model leads to similar behavior for tail latency versus load across all read-write load distributions and request sizes. Although non-linear curve-fitting models can achieve better fit, the accuracy of the linear model has been sufficient for our QoS schedule and we prefer it because of simplicity.

Scheduling Mechanism

The QoS scheduler builds upon the cost model to maintain tail latency and throughput SLOs for LC tenants, while allowing BE tenants to utilize any spare throughput in a fair manner.

Token management: The ReFlex scheduler generates tokens at a rate equal to the maximum weighted IOPS the Flash device can support at a given tail latency SLO. ReFlex enforces the strictest (lowest) latency SLO among all LC tenants that share a Flash device. For example, to serve two tenants with tail read latency SLOs of 500μs and 1ms, respectively, on a Flash device with the cost model shown in Figure 3.3a, the scheduler generates 420K tokens/sec to enforce the 500μs SLO. LC tenants are guaranteed a token supply that satisfies their IOPS SLO, weighted by the read-write ratio indicated in their SLO. For example, assuming 4KB requests and a write cost of 10 tokens, a tenant registering an SLO of 100K IOPS with an 80% read ratio is guaranteed to receive tokens at a rate of $0.8(100\text{K IOPS})(1\ \frac{\text{token}}{\text{I/O}}) + 0.2(100\text{K IOPS})(10\ \frac{\text{tokens}}{\text{I/O}}) = 280\text{K tokens/sec}$.

Algorithm 1 QoS Scheduling Algorithm

```
1: procedure SCHEDULE
2:     time_delta = current_time() − prev_sched_time
3:     prev_sched_time = current_time()
4:     for each latency-critical tenant t do
5:         t.tokens += generate_tokens_LC(t.SLO, time_delta)
6:         if t.tokens < NEG_LIMIT then
7:             notify_control_plane()
8:         while t.demand > 0 and t.tokens > NEG_LIMIT do
9:             t.tokens −= submit_next_req(t.queue)
10:        if t.tokens > POS_LIMIT then
11:            atomic_incr_global_bucket(t.tokens × FRACTION)
12:            t.tokens −= t.tokens × FRACTION
13:    for each best-effort tenant t in round-robin order do
14:        t.tokens += generate_tokens_BE(time_delta)
15:        d = t.demand − t.tokens
16:        if d > 0 then
17:            t.tokens += atomic_decr_global_bucket(d)
18:        t.tokens−=submit_admissible_reqs(t.queue, t.tokens)
19:        if t.tokens > 0 and t.demand == 0 then
20:            atomic_incr_global_bucket(t.tokens)
21:            t.tokens = 0
22:    if all_threads_scheduled() then
23:        atomic_reset_global_bucket()
```

Tokens generated by the scheduler but not allocated to LC tenants are distributed fairly among BE tenants. The scheduler spends a tenant's tokens based on per-request costs as it submits the tenant's requests to the Flash device.

Scheduling Algorithm: Each ReFlex thread enqueues Flash requests in per-tenant, software queues. When the thread reaches the QoS scheduling step in the dataplane execution model (§3.2.1), the thread uses Algorithm 1 to calculate the weighted cost of enqueued requests and submit all admissible requests to the Flash device, gradually spending each tenant's tokens. Depending on the thread load and the batching factor, the execution model enters a scheduling round every 0.5μs to 100μs. The control plane and the batch size limit ensure that the time between scheduler invocations does not exceed 5% of the strictest SLO. Frequent scheduling is necessary to avoid excessive queuing delays and to maintain high utilization of the NVMe device.

Latency-critical tenants: The scheduling algorithm starts by serving LC tenants. First, the scheduler generates tokens for each LC tenant based on their IOPS SLO and the elapsed time since the previous scheduler invocation. Since the control plane has determined each LC tenant's weighted IOPS reservation is admissible, the scheduler can typically submit all queued requests of LC tenants to the NVMe device.

However, since traffic is seldom uniform and tenants may issue more or less IOPS than the average IOPS reserved in their SLO, the scheduler keeps track of each tenant's token usage. We allow LC tenants to temporarily burst above their token allocation to avoid short term queueing. However, we limit the burst size by rate-limiting LC tenants once they reach the token deficit limit

(`NEG_LIMIT`). This parameter is empirically set to -50 tokens to limit the number of expensive write requests in a burst. We also notify the control plane when this limit is reached to detect tenants with incorrect SLOs that need renegotiation.

LC tenants that consume less than their available tokens are allowed to accumulate tokens for future bursts. Accumulation is limited by the `POS_LIMIT` parameter. When reached, the scheduler donates a big fraction of accumulated tokens (empirically 90%) to the global token bucket for use by BE tenants. `POS_LIMIT` is empirically set to the number of tokens the LC tenant received in the last three scheduling rounds to accommodate short bursts without going into deficit.

Best-effort tenants: The scheduler generates tokens for BE tenants by giving each BE tenant a fair share of unallocated throughput on the device. Unallocated device throughput corresponds to the token rate the device can support while enforcing the strictest LC latency SLO minus the sum of LC tenant token rates (based on LC tenant IOPS SLOs). Assuming N BE tenants, every scheduling round, each BE tenant receives $\frac{1}{N}$th of the unallocated token rate times the time elapsed since the previous scheduling round. If a BE tenant does not have enough tokens to submit all of its enqueued requests, the tenant can claim tokens from the global token bucket, which are supplied by LC tenants with spare tokens (if any). BE tenants are scheduled in a round robin order across scheduling rounds to provide fair access to the global token bucket. The scheduler submits a request only if the tenant has sufficient tokens for the request. Rate limiting BE traffic is essential for achieving LC SLOs. Since scheduling rounds occur at high frequency, a typical round may generate only a fraction of a token. BE tenants accumulate tokens over multiple scheduling rounds when their request queues are not empty. When a BE tenant's software queue is empty, we disallow token accumulation to prevent bursting after idle periods. This aspect of the scheduler is inspired by Deficit Round Robin (DRR) scheduling [184]. Tokens left unused by a BE tenant are donated to the global token bucket for use by other BE tenants. To avoid large accumulation allowing BE tenants to issue uncontrolled bursts, we periodically reset the bucket.

If a ReFlex server manages more than one NVMe device, we run an independent instance of the scheduling algorithm for each device with separate token counts and limits. We assume the server machine has sufficient PCIe bandwidth, a condition easily met by PCIe Gen3 systems.

3.3 ReFlex Implementation

ReFlex consists of three components: the server, clients, and control plane. Their implementation leverages open-source code bases.

3.3.1 ReFlex Server

The remote Flash server is the main component of ReFlex. We implemented it as an extension to the open-source, IX dataplane operating system [2, 38]. IX uses hardware support for processor

virtualization (through the Dune module [37]) and multi-queue support in NICs (through the Intel DPDK driver [108]) to gain direct and exclusive access to multiple cores and network queues in a Linux system. These resources are used to run a dataplane kernel and any applications on top of it. The original IX dataplane was developed for network-intensive workloads like in-memory, key-value stores. IX uses run to completion of incoming requests and bounded, adaptive batching to optimize both tail latency and throughput. IX also splits connections between threads, using separate cores and queues to scale without requiring synchronization or significant coherence traffic. These optimizations make IX a great starting point for the ReFlex server.

ReFlex extends IX in the following ways. First, we developed an NVMe driver leveraging Intel's Storage Performance Development Kit (SPDK) [109] to interface to Flash devices and gain exclusive access to NVMe queue pairs. Second, we implemented the dataplane model shown in Figure 3.1. In IX, the run to completion model includes all work for a key-value store request, from packet reception to reply transmission. Directly applying this monolithic run to completion model in ReFlex would require blocking for every Flash access. Instead, we introduce a two step model that retains the efficiency of run to completion but allows for asynchronous access to Flash. The first run to completion step is from packet reception to Flash command submission and the second is from Flash command completion to reply transmission. We maintained adaptive batching with a maximum batch size of 64. Third, we implemented the QoS scheduler as part of the first run to completion step. Fourth, we introduced the system calls and events needed for remote Flash accesses shown in Table 3.1. The original IX defines system calls and events for opening and closing connections, receiving and sending network messages, and managing network errors. We introduced system calls and events to register and unregister tenants, submit and complete NVMe read and write commands, and manage NVMe errors. Finally, we developed the ReFlex user-level server code that consumes events delivered by the dataplane and issues system calls back to it. The cookie parameter allows the user-space server code to track requests and retrieve their context upon an event notification. Note that all event and system calls are communicated over shared memory arrays without the need for blocking, interrupts, or thread scheduling. This also enables batching of systems calls under high load in order to reduce overheads. The dataplane implements zero-copy; buffers for read and write data are initialized in the ReFlex user-space code and provided as a parameter for read and write system calls. They are released after the user-space code is notified of a send completion.

The ReFlex server is written in `C` and consists of the following source lines of code (SLOC): 490 SLOC for the user-level server, 954 SLOC for the IX NVMe driver and 628 SLOC for the dataplane including the QoS scheduler. We leverage code from Intel's DPDK and SPDK and the IX dataplane, including the lwIP TCP stack [73].

Multi-threading operation: ReFlex scales to multiple threads, each using a separate core and separate network and NVMe queues. We parallelize the load by dividing tenants across threads. All

System Calls (batched)		
Type	**Parameters**	**Description**
register	id, latency, IOPS, rw_ratio, cookie	Registers a tenant with SLO
unregister	handle	Unregisters a tenant
read	handle, buf, addr, len, cookie	Read data from Flash into user buf
write	handle, buf, addr, len, cookie	Write data from user buf into flash

Event Conditions		
Type	**Parameters**	**Description**
registered	handle, cookie, status	Registered tenant, or out of resources error
unregistered	handle	Unregistered tenant
response	cookie, status	NVMe read completed
written	cookie, status	NVMe write completed

Table 3.1: Systems calls and event conditions that the ReFlex dataplane adds to IX.

operations across threads are independent and can occur without synchronization, excluding some QoS scheduling actions described in §3.2.2. Specifically, threads need to occasionally synchronize in order to exchange any spare tokens from their LC tenants so that any BE tenant on any thread can benefit from unused Flash bandwidth. Threads use atomic read-modify-write operations to access the global token bucket. The bucket is reset periodically by having each thread asynchronously mark that it has completed at least one scheduling round. The last thread resets the global bucket. This approach avoids locking overheads and decouples QoS scheduling across threads. In particular, it allows threads to perform scheduling at different frequencies, while still maintaining fairness and guaranteeing system-wide SLOs.

Security model: The ReFlex server enforces access control list (ACL) policies at the granularity of tenants and network connections. It checks if a client has the right to open a connection to a specific tenant and if a tenant has read or write permission for an NVMe namespace (range of logical blocks). These checks can be extended to use certificate mechanisms.

Following IX, ReFlex runs its dataplane in protected kernel mode (guest ring 0), while the high-level server code runs in user space (ring 3) as shown in Figure 3.1. Any exploitable bug in parsing remote requests or other high-level server functions cannot lead to loss of hardware control and cannot affect the operation of the dataplane or any ordinary Linux application running on the same machine. This approach allows ReFlex to share Flash devices with other Linux applications. Access to Flash by Linux workloads (kernel or user) is mediated through the protected part of ReFlex that includes the QoS scheduler. From a QoS perspective, Linux requests are treated as latency-critical with specific throughput and latency guarantees. We achieve virtually indistinguishable performance compared to running ReFlex all in user mode. The inherit cost of the kernel to user-mode transition is similar to that of a main memory access [38] and it is compensated for by the improved locality

that ReFlex achieves using run to completion and zero-copy.

Limitations: The current server implementation has some non-fundamental limitations that we will remove in future versions. First, we limit each tenant to using a single ReFlex thread. Since ReFlex can serve up to 850K remote IOPS per thread (§3.4.3) and an application can use multiple tenants to access the same data, this is not a significant bottleneck for any application. In the future, we will load balance connections for individual tenants across threads if their overall demands exceed a single thread's throughput. Second, we have implemented a single networking protocol in the ReFlex dataplane, the ubiquitous TCP/IP [73]. Since TCP/IP is the most heavy-weight protocol used in datacenters, this is a conservative choice that defines a lower bound on ReFlex performance. Both tail latency and throughput will improve when we implement UDP or other, lighter-weight transport protocols. Finally, ReFlex currently serves remote read and write requests without any ordering guarantees, beyond ordering forced by the networking protocol (e.g., order within a TCP connection). In the future, we will support barrier operations that can be used to force ordering and build high-level abstractions like atomic transactions.

3.3.2 ReFlex Clients

Applications can access ReFlex servers over the network using a variety of client approaches. We have implemented two alternatives that represent extreme points in terms of performance.

The first implementation is a user-level library (536 SLOC), similar to the client library for the binary protocol of the memcached key-value store [153]. The library allows applications to open TCP connections to ReFlex and transmit read and write requests to logical blocks. This client approach avoids the performance overheads of the file-system and block layers of the operating system in the client machine. Nevertheless, the client is still subject to any latency or throughput inefficiencies of the networking layer in its operating system (see §3.4).

To support legacy client applications, we also implemented a remote block device driver that exposes a ReFlex server as a Linux block device (845 SLOC). The driver translates conventional Linux block I/O (`bio`) requests to ReFlex accesses issued with the user-level library discussed above. The driver implements the multi-queue (`blk-mq`) kernel API [42] and supports one hardware context per core to enable linear scaling with cores. For each hardware context, the driver opens a socket to the ReFlex server and spawns a kernel thread for receiving and completing incoming responses. To minimize latency, the driver directly issues each block to the server without coalescing as the overhead of ReFlex requests is small (38 bytes per 4KB request) and the bandwidth of NVMe devices does not change significantly if we use requests larger than 4KB. At 4KB, the Linux TCP stack supports up to 70K messages per thread and hence the driver needs to execute at least 4 threads (or 6 for improved latency) to fully utilize a 10GbE interface, as we will show in §3.4.6.

3.3.3 ReFlex Control Plane

The ReFlex control plane consists of two components, a local component that runs on every ReFlex server and a global one. We have currently implemented the former.

The local control plane is responsible for the following actions. First, when a new LC tenant is registered, the control plane determines if the tenant is admissible and which server thread it should be bound to. It uses the strictest latency SLO from all LC tenants and the throughput-latency characteristics of each device to check if the new tenant's SLO can be met without violating SLOs of existing tenants. When a tenant registers or terminates, the control plane re-calculates the rate of token generation for LC and BE tenants. The control plane intervenes if an LC tenant consistently bursts above its SLO allocation by notifying the tenant to renegotiate its SLO. Second, the local control plane monitors the request latency and the thread load. If latency and load are high, it allocates resources for additional threads and rebalances tenants. If load is low, it deallocates threads and their resources, returning them to Linux for general use. This last function is a derivative of the IX control plane that can dynamically rightsize the number and clock frequency of threads used by the IX dataplane without packet loss or reordering [164]. Finally, the control plane periodically calibrates the request cost model and determines the throughput-latency characteristics of each Flash device (see §3.2.2).

In future work, we will develop a global control plane that manages remote Flash resources across a datacenter cluster and optimizes the allocation of Flash capacity and IOPS. For example, the global control plane should try to co-locate tenants with similar tail latency requirements such that strict requirements of one tenant do not limit the IOPS available to other tenants. The global control plane should also maintain global latency and throughput SLOs for applications that span across multiple ReFlex servers [198, 192, 26, 95].

3.4 Evaluation

3.4.1 Experimental Methodology

Hardware setup: Our experimental setup consists of identical server and client machines with Intel Xeon CPU E5-2630 processors (Sandy Bridge EP) with 12 physical cores across two sockets running at 2.3 GHz and 64GB DRAM. The machines use Intel 82599ES 10GbE NICs connected via an Arista 7050S-64 switch. They run Ubuntu LTS 16.04 with a 4.4 Linux kernel. Server machines house PCIe-attached Flash devices, preconditioned with sequential writes to the whole address space followed by a series of random writes to reach steady state performance. We tested ReFlex with three different Flash devices whose request cost models are shown in Figure 3.3. We show results for ReFlex using device A as it achieves the highest raw IOPS (up to 1M IOPS for read-only workloads, see Figure 3.2). For all experiments, we disable power management and operate CPU cores at their

maximum frequency to ensure result fidelity. NICs are configured with jumbo frames enabled and large receive offload (LRO) and generic receive offload (GRO) disabled. LRO and GRO distort unloaded latency as received packets are sometimes buffered instead of being directly delivered to the kernel. We enable interrupt coalescing with a 20μs interval.

Clients: We use Linux-based clients in most experiments. We extend the `mutilate` load generator [133] to use our user-level client library and issue read/write requests to ReFlex. `mutilate` coordinates a large number of client threads across multiple machines to generate a desired throughput while a separate, unloaded client measures latency by issuing one request at a time. To reduce client-side performance overheads, we also evaluate unloaded latency and peak IOPS per core for ReFlex with clients running a similar load generator on top of the IX dataplane, which achieves significantly lower latency and higher throughput than the Linux networking stack.

I/O size: We issue 4KB read and write requests in most experiments. Since we use a 10GbE network infrastructure, clients issuing 4KB IOPS can saturate the NIC of the ReFlex server before they saturate the NVMe Flash device (1M IOPS peak). Hence, we use 1KB requests in some experiments to stress IOPS of the ReFlex server. Modern datacenters include 40GbE networking infrastructure and future datacenters will likely deploy 100GbE. Both technologies will remove this bottleneck.

Baseline: We compare the performance of remote accesses over ReFlex to that of issuing local accesses to the Flash device using SPDK [109]. SPDK offers the best local performance we can expect as it gives software direct access to NVMe queues without the need to go through the Linux filesystem or block device layers. We also compare ReFlex to two software-based schemes for remote Flash access: 1) the Linux iSCSI system [155] and 2) a lightweight remote storage server that maximizes performance on Linux by efficiently handling multiple connections per thread using `libevent` and overlapping communication and computation using `libaio`. We do not have access to a hardware-accelerated remote Flash environment, but we compare to results quoted in a public presentation on NVMe over RDMA Fabrics [143]. In §3.4.4, we evaluate the performance problems that arise when NVMe devices are shared without a software-based QoS scheduler like the one in ReFlex.

3.4.2 Unloaded latency

We first measure unloaded latency for Flash accesses. Table 3.2 shows the average and 95$^{\text{th}}$ percentile latency of 4KB random read and write requests issued with queue depth 1. Remote accesses include the round-trip networking overheads in both client and server. Remote access over iSCSI increases latency by 2.8× for read requests due to heavy-weight protocol processing on both the client and server side, involving data copying between socket, SCSI and application buffers. The `libaio-libevent` remote Flash server is significantly faster than iSCSI, but the Linux network and storage stacks still add over 100μs to average and tail latency. The dataplane execution model

	Reads (μs)		**Writes** (μs)	
	Avg	p95	Avg	p95
Local (SPDK)	78	90	11	17
iSCSI	211	251	155	215
Libaio (Linux Client)	183	205	180	205
Libaio (IX Client)	121	139	117	144
ReFlex (Linux Client)	117	135	58	64
ReFlex (IX Client)	99	113	31	34

Table 3.2: Unloaded Flash read and write latency for 4KB random I/Os, comparing local access latency and remote access using baseline systems (iSCSI and liabio) and ReFlex. Remote access latency includes round-trip network latency for client and server.

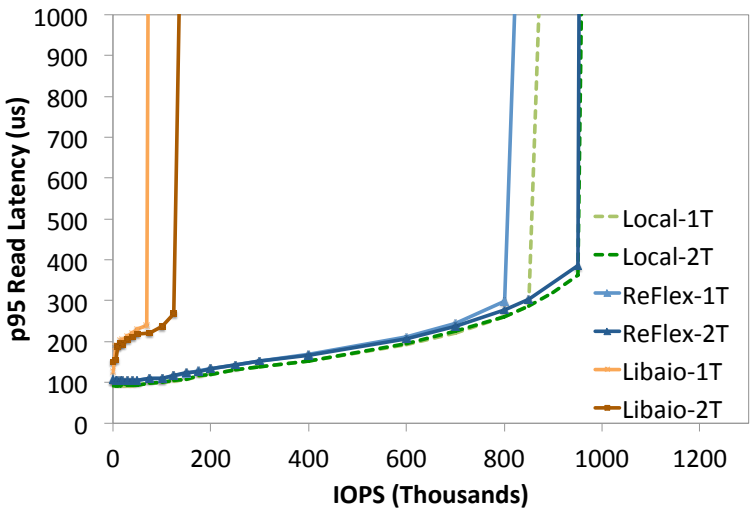


Figure 3.4: Tail read latency vs. throughput for 1KB read-only requests, comparing local and remote access with ReFlex and Libaio baseline. 1T and 2T refer to the use of 1 and 2 cores on the remote Flash storage server, respectively.

of ReFlex adds 21μs to local Flash latency (IX client). At 113μs of tail read latency, a ReFlex server is close to the performance of many (local) NVMe devices. Unloaded write latency is lower than read latency due to DRAM buffering on the Flash device. Thus, the overhead of iSCSI and `libaio-libevent` is even more significant for write I/Os. ReFlex outperforms both iSCSI and `libaio-libevent`, adding 20μs to local write latency (IX client). Comparing ReFlex latency results with Linux and IX clients shows that for low latency remote Flash access, it is also important to optimize the client.

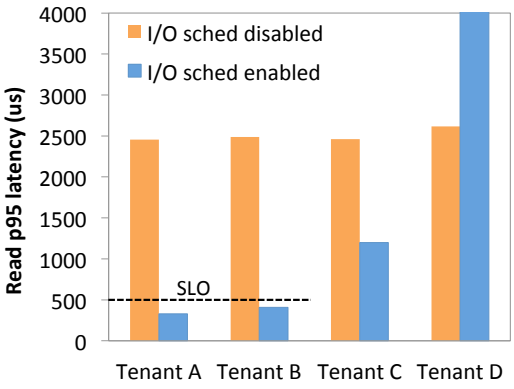
NVMe over Fabrics provides marginally less latency overhead (8μs), measured with a higher throughput 40GbE Chelsio NIC (lower transmit latency for 4KB) and a 3.6GHz Haswell CPU (versus a 2.3GHz Sandy Bridge CPU) [143]. Remote Flash latency with ReFlex includes a full TCP/IP stack and a QoS scheduler that allows multiple clients to connect to the Flash server. ReFlex would likely benefit from TCP offloading in Chelsio NICs.

3.4.3 Throughput and CPU Resource Cost

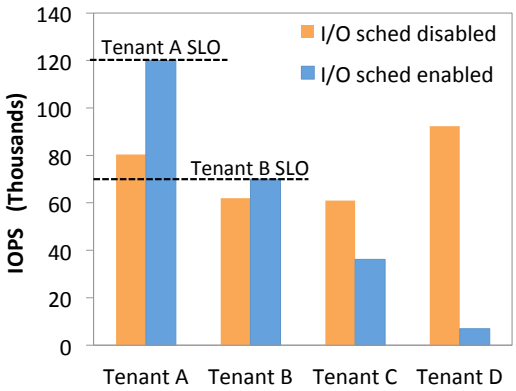
Figure 3.4 plots tail latency (95^{th} percentile) as a function of throughput (IOPS) for 1KB read-only requests. Even for local accesses to Flash with SPDK, it takes two cores to saturate the 1M IOPS of the Flash device. A single core can support up to 870K IOPS on local Flash. ReFlex achieves up to 850K IOPS with a single core for network and storage processing. With two cores, ReFlex saturates 1M IOPS on Flash, introducing negligible latency overhead compared to local access. In contrast, the `libaio-libevent` server achieves only 75K IOPS/core and at higher latency due to higher compute intensity for request processing. This server requires over 10× more CPU cores to achieve the throughput of ReFlex. The hardware-accelerated NVMe over Fabrics can reportedly achieve 460K IOPS at 20% utilization of a 3.6GHz Haswell core [143].

At high load, a ReFlex thread spends about 20% of execution time on TCP/IP processing. This indicates that, coupled with a lighter network protocol, ReFlex can deliver even higher throughput. The time spent on QoS scheduling varies between 2% and 8%, depending on the number of tenants served.

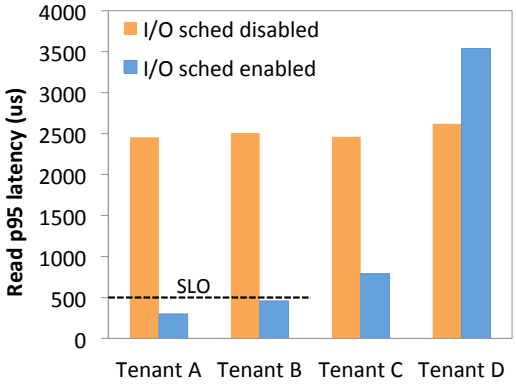
ReFlex's ability to serve millions of IOPS with a small number of cores without impacting tail latency is important for making remote Flash practical and cost effective in datacenters. To put IOPS per core into perspective, assume we deploy ReFlex on the latest Broadwell or Skylake class CPUs by Intel. The improved core performance will likely allow ReFlex to reach 1M IOPS/core. Assuming 20 cores per CPU socket, ReFlex will be able to share a 1M IOPS Flash device using 2.5% of the compute capacity of a 2-socket server. Alternatively, using 4 Flash devices, ReFlex will need 8% of the server's compute capacity to saturate a 100GbE link with 4KB I/Os.



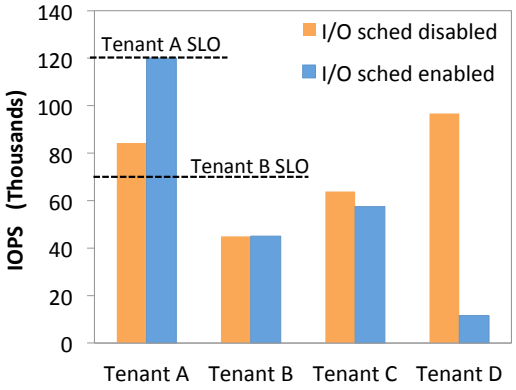
(a) Scenario 1: p95 latency



(b) Scenario 1: IOPS



(c) Scenario 2: p95 latency



(d) Scenario 2: IOPS

Figure 3.5: Tail latency and IOPS for 4 tenants sharing a ReFlex server, comparing performance with and without the QoS scheduler enabled. Tenants A and B are LC, while tenants C and D are BE. Tenants issue 4kB I/Os with read ratios of 100%, 80%, 95%, and 25%, respectively. In Scenario 1, tenants A and B attempt to use all the IOPS in their SLO. In Scenario 2, tenant B uses less than its reservation.

3.4.4 Performance QoS and Isolation

We now evaluate the QoS scheduler using multiple tenants with different SLOs. The following experiments use a single ReFlex thread. We evaluate multi-core scalability in §3.4.5.

We first consider Scenario 1 (Figure 3.5a - 3.5b), where two latency-critical (A, B) and two best-effort (C, D) tenants share a ReFlex server. Both A and B require 95^{th} percentile read latency of 500μs. Tenant A requires 120K IOPS at 100% read, while B requires 70K IOPS at 80% read. C and D are best effort tenants with 95% and 25% read loads, respectively. To guarantee read tail latency below the 500μs SLO, our Flash device can support up to 420K weighted IOPS. Thus, the QoS scheduler generates 420K tokens/sec. LC tenant A receives 120K tokens/sec while tenant B receives 196K tokens/sec = 0.8(70K IOPS)(1 $\frac{\text{token}}{\text{I/O}}$) + 0.2(70K IOPS)(10 $\frac{\text{tokens}}{\text{I/O}}$). Thus, the two LC tenants collectively reserve 75% of the device throughput, leaving 25% of tokens for BE tenants. Figure 3.5 shows the tail latency and IOPS for each tenant with the QoS scheduler disabled and enabled. Without QoS scheduling, tail read latency is above 2ms for all tenants due to read/write interference. Tenant B also operates below its SLO throughput. With QoS scheduling enabled, latency and throughput SLOs are met for both LC tenants (Figure 3.5a) at the expense of BE throughput (Figure 3.5b). BE tenants C and D receive a fair share of unallocated tokens (52K tokens/sec each), but D achieves lower IOPS than C due to its higher percentage of write I/Os (writes cost 10 times more tokens than reads).

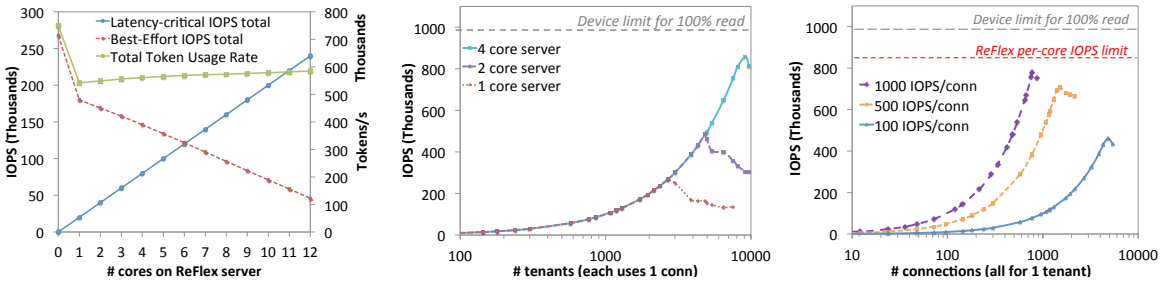
Scenario 2 uses the same tenants as Scenario 1 with identical SLOs. However, latency-critical tenant B issues only 45K IOPS instead of the 70K reserved in its SLO. BE tenants can now reach higher throughput (Figure 3.5c - 3.5d), as they acquire the unused tokens of tenant B, in addition to tokens not allocated to LC tenants. The round-robin serving of BE tenants ensures fair access to unused tokens.

While these scenarios involve just 4 tenants, they are sufficient to show the need for QoS scheduling for remote Flash accesses, beyond what hardware provides. Our QoS scheduler can guarantee SLOs while being work-conserving and fair for best-effort tenants.

3.4.5 Scalability

We now evaluate how ReFlex scales in the dimensions of cores, tenants, and connections.

Cores: We run ReFlex with up to 12 cores (6 cores per socket) to test the scheduler's multi-core scalability. Each thread manages a single LC tenant with an SLO of 20K IOPS (90% read, 4KB requests) at up to 2ms tail read latency (95^{th} percentile). The 2ms latency SLO allows our Flash device to serve up to 12 such tenants before the SLO is no longer admissible due to too much write interference. Two ReFlex threads also each serve a BE tenant (80% read, 4KB). Figure 3.6a shows a linear increase in the aggregate IOPS for LC tenants as we scale the number of cores (tenants) without any scaling bottleneck in the scheduler. Meanwhile, aggregate BE IOPS decrease due to rate-limiting with less spare bandwidth on the device. Although not shown in Figure 3.6, the tail



(a) Multi-core scaling (b) Tenant scaling (c) Connection scaling

Figure 3.6: Scalability experiments. In Fig 3.6a, ReFlex scales to 12 cores, enforcing a 2ms latency SLO for 90% read (LC) and 80% read (BE) tenants while maintaining high Flash utilization (570K tokens/s). Fig 3.6b shows a single ReFlex core can support up to 2,500 tenants. Fig 3.6c shows a single ReFlex core can serve thousands of TCP connections.

read latency of all LC tenants stays below the 2ms SLO. The total token usage rate (green line with values marked on the secondary y-axis) is high when no LC tenants are registered since the two BE tenants are allowed to issue as many requests as the device can handle. As soon as the first LC tenant registers, the scheduler caps the token rate to 570K tokens/s to enforce the 2ms SLO. Token usage remains at this level as we scale the number of cores, as ReFlex saturates the Flash device at all points without violating SLOs.

Tenants: We evaluate the number of tenants each ReFlex thread can serve before tenant management becomes a performance bottleneck. Each tenant uses a single connection to issue 100 1KB read IOPS. In this experiment, low IOPS per connection are necessary to avoid saturating the Flash device before reaching a tenant scaling limit. Figure 3.6b shows that a single ReFlex core can serve up to 2,500 tenants, while 2 ReFlex cores serve 5,000 tenants, and a 4-core ReFlex server comes close to supporting 10K tenants, at which point we approach the 1M read-only IOPS limit of the Flash device. As we scale the number of tenants per thread, tail latency may increase as the QoS scheduling frequency decreases. This case is detected by the ReFlex control plane which allocates more cores and rebalances tenants as needed (see §3.3.3).

TCP Connections: A tenant may be used to track the QoS requirement of an application that uses multiple client machines and threads. Hence, it is important to know how many TCP connections the ReFlex server can handle. Figure 3.6c plots the throughput of a single ReFlex thread when scaling the number of TCP connections associated with a tenant. At 100 IOPS per connection, a single ReFlex thread can support up to 5K connections. Performance degrades beyond this point as the TCP connection state no longer fits in the last-level cache and TCP/IP processing slows down due to main memory accesses. This is similar to the connection scaling behavior of IX [38], which saturates at 10K connections in experiments with smaller messages (64B vs. 1KB) which trigger fewer misses. With 1K IOPS/connection, the ReFlex core approaches its peak bandwidth, achieving

780K IOPS with 850 connections. The peak bandwidth is lower than the 850K IOPS in §3.4.3 due to higher cache pressure.

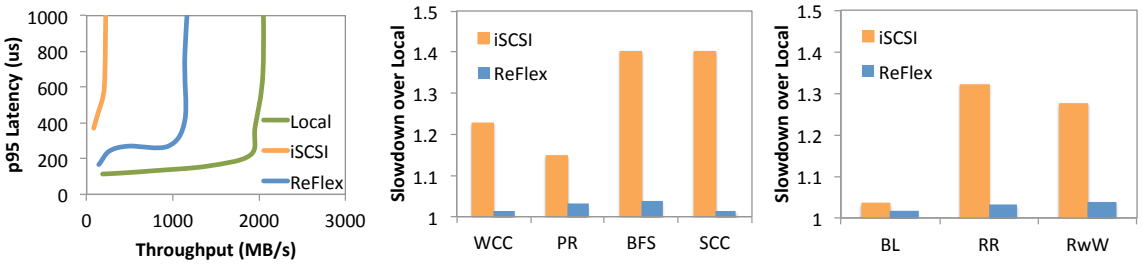
3.4.6 Linux Application Performance

We now use the remote block device driver for ReFlex to evaluate performance with legacy Linux applications. We compare to performance with the local NVMe device driver and Linux iSCSI remote block I/O. We show results for the following applications: the flexible I/O tester (FIO) [111], the FlashX graph analytics framework [221], and Facebook's RocksDB key-value store [75].

FIO: Figure 3.7a shows the latency-throughput curves for FIO issuing 4KB random reads with queue depth up to 64. We need multiple FIO threads to reach maximum throughput; 5 threads with the local NVMe driver, 3 threads with iSCSI, and 6 with the ReFlex block driver. As expected, ReFlex stops scaling when it saturates the 10GbE network interfaces at both the client and the server. However, since FIO on top of ReFlex scales linearly up to 6 threads, we expect it will be able to match local throughput given higher bandwidth network links. The higher latency of ReFlex in this experiment is due to the client-side overheads of the Linux block and networking layers. Still, ReFlex provides 4× higher throughput than iSCSI and 2× lower tail and average latency. We evaluated ReFlex with optimized clients in §3.4.3.

FlashX: We use FlashX, a graph processing framework that uses the SAFS user-space filesystem to efficiently store and retrieve vertex and edge data from Flash. We execute four graph benchmarks including weakly connected components (WCC), pagerank (PR), breadth-first search (BFS) and strongly connected components (SCC) on the SOC-LiveJournal1 social network graph from the SNAP dataset [131]. The graph contains 4.8M vertices and 68.9M edges, which we store on a local block device or on a remote block device through iSCSI or ReFlex. Figure 3.7b shows the impact of accessing remote Flash on the end-to-end application execution time. Compared to performance on local Flash, ReFlex introduces only a small slowdown, between 1% for WCC and 3.8% for BFS. In contrast, iSCSI reduces performance by 15% for PR and up to 40% for BFS and SCC.

RocksDB: Finally, we use RocksDB to evaluate key-value store performance on Flash with ReFlex. We install an ext4 filesystem on the NVMe block device and mount it as either local or remote via ReFlex or iSCSI. We place both RocksDB's database and its write-ahead-log on Flash. We generate a workload using `db_bench`, a benchmarking tool provided with RocksDB. We use `cgroups` [141] to limit memory and reduce the effect of Linux's page cache, thus exercising Flash storage with a short experiment on a 43GB database. We first populate the database with the *bulkload* (BL) routine and then execute the *randomread* (RR) and *readwhilewriting* (RwW) benchmarks. Figure 3.7c shows the end-to-end execution time slowdown of RocksDB over iSCSI and ReFlex compared to local Flash. For the write-heavy BL test, performance is almost equal between local and remote as the Flash itself limits IOPS. For RR and RwW, iSCSI shows a slow down of 1.32× and 1.27×, respectively, while ReFlex slows down performance by less than 4%.



(a) FIO p95 Access Latency (b) FlashX Performance (c) RocksDB

Figure 3.7: Application performance with Linux block device driver for ReFlex vs. iSCSI baseline.

3.5 Discussion

There are two limitations of current Flash hardware that are particularly relevant to ReFlex.

Read/write interference: Write operations have a big impact on the tail latency of concurrent read requests. Our scheduler uses a request cost model to avoid pushing beyond the latency-throughput capabilities of the Flash device for the current read/write ratio. However, we are still limited to enforcing tail read latency SLOs at the 95^{th} percentile. Stricter SLOs, such as 99^{th} or 99.9^{th} percentile are difficult to enforce on existing Flash devices without dramatically reducing IOPS as reads frequently stall behind writes, garbage collection, or wear leveling tasks. Future Flash devices should limit read/write interference, targeting tail behavior in addition to average. For example, the Flash Translation Layer (FTL) could always read out and buffer Flash pages before writing in them [219, 13].

Hardware support for request scheduling: Existing Flash devices schedule requests from different NVMe hardware queues using simplistic round-robin arbitration. To guarantee SLOs, ReFlex has to use a software scheduler that implements rate limiting and priorities. The NVMe specification defines a weighted round-robin arbiter [154], but it is not implemented by any Flash device we are aware of. This arbiter would allow ReFlex threads to submit requests to hardware queues with properly weighted priorities, thus eliminating the need to enforce priorities between tenants in software. ReFlex would still implement rate limiting in software as it must manage the device latency-throughput characteristics under varying read/write ratios, defend against SLO violations (long bursts by LC tenants), and support a number of tenants that may exceed the number of hardware queues.

3.6 Related Work

We discuss related work on storage QoS and high performance network stacks. Alternative approaches for remote access to Flash were discussed in Section 2.3.1.

Storage QoS: Prior work has extensively studied quality of service and fairness for shared storage [27, 91, 142, 94, 186, 218, 92]. Timeslice-based I/O schedulers like Argon, CFQ, and FIOS offer tenants exclusive device access for regulated time quanta to achieve fairness [207, 33, 159]. This approach can lead to poor responsiveness and timeslices may not always be fair on Flash as background tasks (i.e., garbage collection) impact device performance.

In contrast, fair-queuing-based solutions interleave requests from all tenants. The original weighted fair queuing schedulers [70, 158] have successfully been adapted from network to storage I/O with support for reordering, throttling, and/or batching requests to leverage device parallelism [183, 46, 114, 181, 203, 93]. Our I/O scheduler resembles Deficit Round Robin in that tenants accumulate tokens each round, so long as they have demand [184]. Zygaria and AQuA also apply a token bucket approach to serve real-time tenants while offering spare device bandwidth to best-effort traffic, but they only provide throughput guarantees while ReFlex also enforces tail latency SLOs [212, 213].

Tail latency SLOs: PriorityMeister provides tail latency guarantees even at the 99.99^{th} percentile by mediating access to shared network and storage resources using a token-bucket mechanism similar to ReFlex [223]. Unlike ReFlex, the scheduler profiles workloads to assign different priorities to latency-critical tenants. Cake uses a feedback controller to enforce tail latency SLOs, but only supports a single latency-critical tenant [209]. Avatar relies on feedback instead of device-specific performance models to control tail latency on disk [218].

Flash-specific challenges: Many I/O schedulers specifically designed for Flash use a request cost model to account for read/write interference. FIOS was one of the first schedulers to address Flash write interference and provide fairness using timeslices [159]. FlashFQ, a virtual-time based scheduler, improves fairness and responsiveness through throttled dispatch and I/O anticipation [181]. Libra is an I/O scheduling framework that allocates per-tenant throughput reservations and uses a virtual IOPS metric to capture the non-linearity between raw IOPS and bandwidth [185]. While FIOS, FlashFQ and Libra all assign I/O costs, their cost models do not necessarily capture a request's impact on the *tail latency* of concurrent I/Os, since these schedulers are designed for fairness and throughput guarantees rather than latency QoS.

High Performance Networking: ReFlex leverages the IX dataplane for high performance networking [38]. Alternative network stacks, such as mTCP [113], Sandstorm [139] and OpenOnload [188], apply similar techniques in user space to achieve high throughput and/or low latency networking.

3.7 Conclusion

We described ReFlex, a software system for remote Flash access over commodity networking. ReFlex uses a dataplane design to closely integrate and reduce the overheads of networking and storage processing. This allows the system to serve up to 850K IOPS per core while adding only 21μs over direct access to local Flash. The QoS scheduler in ReFlex enforces latency and throughput SLOs across thousands of tenants sharing a device. ReFlex allows applications to flexibly allocate Flash across any machine in the datacenter and still achieve nearly identical performance to using local Flash.

Chapter 4

Pocket: Elastic Ephemeral Storage for Serverless Analytics

4.1 Introduction

Serverless computing is becoming an increasingly popular cloud service due to its high elasticity and fine-grain billing. Serverless platforms like AWS Lambda, Google Cloud Functions, and Azure Functions enable users to quickly launch thousands of light-weight tasks, as opposed to entire virtual machines. The number of serverless tasks scales automatically based on application demands and users are charged only for the resources their tasks consume, at millisecond granularity [23, 87, 145].

While serverless platforms were originally developed for web microservices and IoT applications, their elasticity and billing advantages make them appealing for data intensive applications such as interactive analytics. As discussed in Section 2.2.3, several recent frameworks launch large numbers of fine-grain tasks on serverless platforms to exploit all available parallelism in an analytics job and achieve near real-time performance [78, 116, 64]. In contrast to traditional serverless applications that consist of a single function executed when a new request arrives, analytics jobs typically consist of multiple stages and require sharing of state and data across stages of tasks (e.g., data shuffling).

Most analytics frameworks (e.g., Spark) implement data sharing with a long-running framework agent on each node buffering intermediate data in local storage [217]. This enables tasks from different execution stages to directly exchange intermediate data over the network. However, in serverless deployments, there is no long-running application framework agent to manage local storage. Furthermore, serverless applications have no control over task scheduling or placement, making direct communication among tasks difficult. As a result of these limitations, the natural approach for data sharing in serverless applications is to use a remote storage service. For instance, early frameworks for serverless analytics either use object stores (e.g., S3 [22]), databases (e.g., CouchDB [3]) or distributed caches (e.g., Redis [128]).

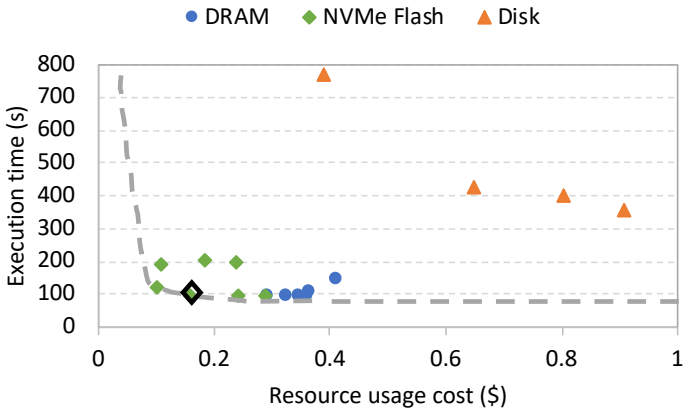


Figure 4.1: Performance-cost trade-off for a video analytics job with various ephemeral storage cluster configurations.

Unfortunately, existing storage services are not a good fit for sharing short-lived intermediate data in serverless applications. We refer to the intermediate data as *ephemeral data* to distinguish it from input and output data which requires long-term storage. File systems, object stores and NoSQL databases prioritize providing durable, long-term, and highly-available storage rather than optimizing for performance and cost. Distributed key-value stores offer good performance, but burden users with managing the storage cluster scale and configuration, which includes selecting the appropriate compute, storage and network resources to provision.

The availability of different storage technologies (e.g., DRAM, NVM, Flash, and HDD) increases the complexity of finding the best cluster configuration for performance and cost. However, the choice of storage technology is critical since jobs may exhibit different storage latency, bandwidth and capacity requirements while different storage technologies vary significantly in terms of their performance characteristics and cost [123]. As an example, Figure 4.1 plots the performance-cost trade-off for a serverless video analytics application using a distributed ephemeral data store configured with different storage technologies, number of nodes, compute resources per node, and network bandwidth (see §4.6.1 for our AWS experiment setup). Each resource configuration leads to different performance and cost. Finding Pareto efficient storage allocations for a job is non-trivial and gets more complicated with multiple jobs.

We present *Pocket*,[1] a distributed data store designed for efficient data sharing in serverless analytics. Pocket offers high throughput and low latency for arbitrary size data sets, automatic resource scaling, and intelligent data placement across multiple storage tiers such as DRAM, Flash, and disk. The unique properties of Pocket result from a strict separation of responsibilities across three planes: a control plane which allocates storage resources for jobs, a metadata plane which manages distributed data placement, and a 'dumb' (i.e., metadata-oblivious) data plane responsible

[1]Pocket is open-source software. The code is available at: `https://github.com/stanford-mast/pocket`.

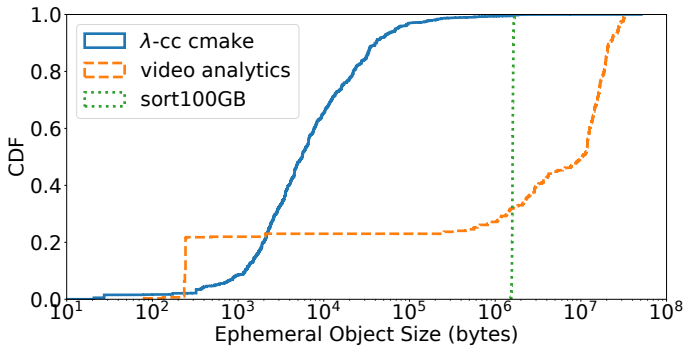


Figure 4.2: Ephemeral object size CDF. Objects are 100s of bytes to 100s of MBs.

for storing data. Pocket scales all three planes independently at fine resource and time granularity based on the current load. Pocket uses heuristics, which take into account job characteristics, to allocate the right storage media, capacity, bandwidth and CPU resources for cost and performance efficiency. The storage API exposes deliberately simple I/O operations for sub-millisecond access latency. We intend for Pocket to be managed by cloud providers and offered to users with a pay-what-you-use cost model.

We deploy Pocket on Amazon EC2 and evaluate the system using using three serverless analytics workloads: video analytics, MapReduce sort, and distributed source code compilation. We show that Pocket is capable of rightsizing the type and number of resources such that jobs achieve similar performance compared to using ElastiCache Redis, a DRAM-based key-value store, while saving almost 60% in cost.

4.2 Storage for Serverless Analytics

Early work in serverless analytics has identified the challenge of storing and exchanging data between hundreds of fine-grain, short-lived tasks [116, 78]. We build on the study of ephemeral storage requirements for serverless analytics applications in Section 2.2.3 to synthesize essential properties for an ephemeral data storage solution. We also discuss why current systems are not able to meet the ephemeral I/O demands of serverless analytics applications.

Our focus is on ephemeral data as the original input and final output data of analytics jobs typically has long-term availability and durability requirements that are well served by the variety of file systems, object stores, and databases available in the cloud.

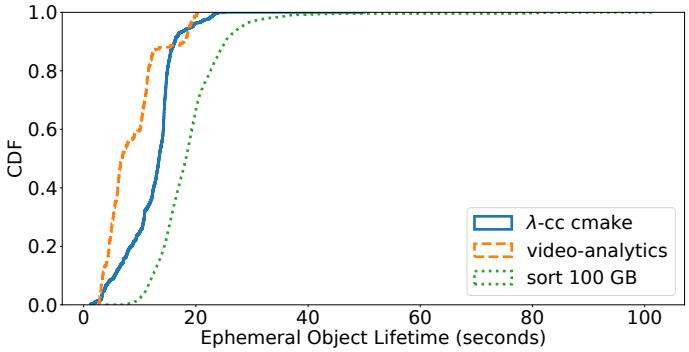


Figure 4.3: Ephemeral object lifetime CDF. Objects have short lifetime.

4.2.1 Ephemeral Storage Requirements

High performance for a wide range of object sizes: Serverless analytics applications vary considerably in the way they store, distribute, and process data. This diversity is reflected in the granularity of ephemeral data that is generated during a job. Figure 4.2 shows the ephemeral object size distribution for a distributed lambda compilation of the `cmake` program, a serverless video analytics job using the Thousand Island (THIS) video scanner [167], and a 100 GB MapReduce sort job on lambdas. The key observation is that ephemeral data access granularity varies greatly in size, ranging from hundreds of bytes to hundreds of megabytes. We observe a straight line for sorting as its ephemeral data size is equal to the partition size. However, with a different dataset size and/or number of workers, the location of the line changes. Applications that read/write large objects demand high throughput (e.g., we find that sorting 100 GB with 500 lambdas requires up to 7.5 GB/s of ephemeral storage throughput) while low latency is important for small object accesses. *Thus, an ephemeral data store must deliver high bandwidth, low latency, and high IOPS for the entire range of object sizes.*

Automatic and fine-grain scaling: One of the key promises of serverless computing is agility to dynamically meet application demands. Serverless frameworks can launch thousands of short-lived tasks instantaneously. Thus, an ephemeral data store for serverless applications can observe a storm of I/O requests within a fraction of a second. Once the load dissipates, the storage (just like the compute) resources should be scaled down for cost efficiency. Scaling up or down to meet elastic application demands requires a storage solution capable of growing and shrinking in multiple resource dimensions (e.g., adding/removing storage capacity and bandwidth, network bandwidth, and CPU cores) at a fine time granularity on the order of seconds. In addition, users of serverless platforms desire a storage service that automatically manages resources and charges users only for the fine-grain resources their jobs actually consume, to match the abstraction that serverless computing already provides for compute and memory resources. Automatic resource management is important since navigating cluster configuration performance-cost trade-offs is a burden for users. For example,

finding the Pareto optimal point outlined in Figure 4.1 is non-trivial; it is the point beyond which adding resources only increases cost without improving execution time while using any lower-cost resource allocation results in sub-optimal execution time. In summary, *an ephemeral data store must automatically rightsize resources to satisfy application I/O requirements while minimizing cost.*

Storage technology awareness: In addition to rightsizing cluster resources, the storage system also needs to decide which storage technology to use for which data. The variety of storage media available in the cloud allow for different performance-cost trade-offs, as shown in Figure 4.1. Each storage technology differs in terms of I/O latency, throughput and IOPS per GB of capacity, and the cost per GB. The optimal choice of storage media for a job depends on its characteristics. *Hence, the ephemeral data store must place application data on the right storage technology tier(s) for performance and cost efficiency.*

Fault-(in)tolerance: Typically a data store must deal with failures while keeping the service up and running. Hence, it is common for storage systems to use fault-tolerance techniques such as replication and erasure codes [104, 177, 120]. For data that needs to be stored long-term, such as the original input and final output data for analytics workloads, the cost of data unavailability typically outweighs the cost of fault-tolerance mechanisms. However, as shown in Figure 4.3, ephemeral data has a short lifetime of 10-100s of seconds. Unlike the original input and final output data, ephemeral data is only valuable during the execution of a job and can easily be regenerated.

Furthermore, fault tolerance is typically baked into compute frameworks, such that storing the data and computing it become interchangeable [96]. For example, Spark uses a data abstraction called resilient distributed datasets (RDDs) to mitigate stragglers and track lineage information for fast data recovery [217]. *Hence, we argue that an ephemeral storage solution does not have to provide high fault-tolerance as expected of traditional storage systems.*

4.2.2 Existing Systems

Existing storage systems do not satisfy the combination of requirements outlined in § 4.2.1. We describe different categories of systems and summarize why they fall short for elastic ephemeral storage in Table 4.1.

Serverless applications commonly use fully-managed cloud storage services, such as Amazon S3, Google Cloud Storage, and DynamoDB. These systems extend the 'serverless' abstraction to storage, charging users only for the capacity and bandwidth they use [22, 66]. While such services automatically scale resources based on usage, they are optimized for high durability hence their agility is limited and they do not meet the performance requirements of serverless analytics applications. For example, DynamoDB only supports up to 400KB sized objects while S3 has high latency overhead (e.g., a 1 KB read takes ∼12 ms) and insufficient throughput for highly parallel applications. For example, sorting 100 GB with 500 or more workers results in request rate limit errors when S3 is used for intermediate data. Furthermore, although users pay only $0.005 per access for S3, storing

	Elastic scaling	**Latency**	**Throughput**	**Max object size**	**Cost**
S3	Auto, coarse-grain	High	Medium	5 TB	$
DynamoDB	Auto, fine-grain, pay per hour	Medium	Low	400 KB	$$
Elasticache Redis	Manual	Low	High	512 MB	$$$
Aerospike	Manual	Low	High	1 MB	$$
Apache Crail	Manual	Low	High	any size	$$
Desired for λs	*Auto, fine-grain, pay per second*	*Low*	*High*	*any size*	*$*

Table 4.1: Comparison of existing storage systems and desired properties for ephemeral storage in serverless analytics.

ephemeral data in S3 can become prohibitively expensive as we scale the job input data size and number of workers, due to the large number of shuffle data file that need to be written and read [166].

In-memory key-value stores, such as Redis and Memcached, provide another option for storing ephemeral data [128, 10]. These systems offer low latency and high throughput but at the higher cost of DRAM. They also require users to manage their own storage instances and manually scale resources. Although Amazon and Azure offer managed Redis clusters through their ElastiCache and Redis Cache services respectively, they do not automate storage management as desired by serverless applications [19, 146]. Users must still select instance types with the appropriate memory, compute and network resources to match their application requirements. In addition, changing instance types or adding/removing nodes can require tearing down and restarting clusters, with nodes taking minutes to start up while the service is billed for hourly usage.

Another category of systems use Flash storage to decrease cost, while still offering good performance. For example, Aerospike is a popular Flash-based NoSQL database [191]. Alluxio/Tachyon is designed to enable fast and fault-tolerant data sharing between multiple jobs [134]. Apache Crail is a distributed storage system that uses multiple media tiers to balance performance and cost [4]. Unfortunately, users must manually configure and scale their storage cluster resources to adapt to elastic job I/O requirements. Finding Pareto optimal deployments for performance and cost efficiency is non-trivial, as illustrated for a single job in Figure 4.1. Cluster configuration becomes even more complex when taking into account the requirements of multiple overlapping jobs.

4.3 Pocket Design

We introduce *Pocket*, an elastic distributed storage service for ephemeral data that automatically and dynamically rightsizes storage cluster resource allocations to provide high I/O performance while minimizing cost. Pocket addresses the requirements outlined in §4.2.1 by applying the following key design principles:

1. **Separation of responsibilities:** Pocket divides responsibilities across three different planes: the control plane, the metadata plane, and the data plane. The control plane manages cluster sizing and data placement. The metadata plane tracks the data stored across nodes in the data plane. The three planes can be scaled independently based on variations in load, as described in §4.4.2.

2. **Sub-second response time:** All I/O operations are deliberately simple, targeting sub-millisecond latencies. Pocket's storage servers are optimized for fast I/O and are only responsible for storing data (not metadata), making them simple to scale up or down. The controller scales resources at second granularity and balances load by intelligently steering incoming job data. This makes Pocket elastic.

3. **Multi-tier storage:** Pocket leverages different storage media (DRAM, Flash, disk) to store a job's data in the tier(s) that satisfy the I/O demands of the application while minimizing cost (see §4.4.1).

4.3.1 System Architecture

Figure 4.4 shows Pocket's system architecture. The system consists of a logically centralized controller, one or more metadata servers, and multiple data plane storage servers.

The controller, which we describe in §4.4, allocates storage resources for jobs and dynamically scales Pocket metadata and storage nodes up and down as the number of jobs and their requirements vary over time. The controller also makes data placement decisions for jobs (i.e., which nodes and storage media to use for a job's data).

Metadata servers enforce coarse-grain data placement policies generated by the controller by steering client requests to appropriate storage servers. Pocket's metadata plane manages data at the granularity of *blocks*, whose size is configurable. We use a 64 KB block size in our deployment. Objects larger than the block size are divided into blocks and distributed across storage servers, enabling Pocket to support arbitrary object sizes. Clients access data blocks on metadata-oblivious, performance-optimized storage servers equipped with different storage media, such as DRAM, Flash, and/or HDD.

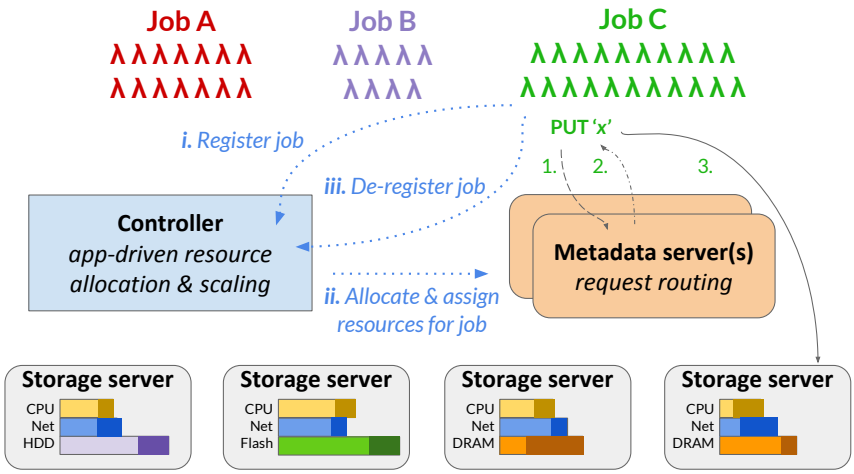


Figure 4.4: Pocket system architecture and the steps to register job C, issue a PUT from a lambda and de-register the job. The colored bars on storage servers show used and allocated resources for all jobs in the cluster.

4.3.2 Application Interface

Table 4.2 outlines Pocket's application interface. Pocket exposes an object store API with additional functions tailored to the ephemeral storage use-case. We describe these functions and how they map to Pocket's separate control, metadata and data planes.

Control functions: Applications use two API calls, `register_job` and `deregister_job`, to interact with the Pocket controller. The `register_job` call accepts hints about a job's characteristics (e.g., degree of parallelism, latency-sensitivity) and requirements (e.g., capacity, throughput). These optional hints help the controller rightsize resource allocations to optimize performance and cost (see §4.4.1). The `register_job` call returns a job identifier and the metadata server(s) assigned for managing the job's data. The `deregister_job` call notifies the controller that a serverless job has completed.

Metadata functions: While control API calls are issued once per job, serverless tasks in a job can interact with Pocket metadata servers multiple times during their lifetime to write and read ephemeral data. Serverless clients use the `connect` call to establish a connection with Pocket's metadata service. Data in Pocket is stored in objects which are organized in buckets. Objects and buckets are identified using names (strings). Clients can create and delete buckets and enumerate objects in buckets by passing their job identifier and the bucket name. Clients can also lookup and delete existing objects. These metadata operations are similar to those supported by other object stores like Amazon S3.

In our current design, Pocket stores all of a job's data in a top-level bucket identified by the job's ID, which is created during job registration by the controller. This implies each job is assigned

Client API Function	Description
register_job(jobname, hints=None)	register job with controller and provide optional hints, returns a job ID and metadata server IP address
deregister_job(jobid)	notify controller job has finished, delete job's non-PERSIST data
connect(metadata_server_address)	open connection to metadata server
close()	close connection to metadata server
create_bucket(jobid, bucketname)	create a bucket
delete_bucket(jobid, bucketname)	delete a bucket
enumerate(jobid, bucketname)	enumerate objects in a bucket
lookup(jobid, obj_name)	return `true` if obj_name data exists, else `false`
delete(jobid, obj_name)	delete data
put(jobid, src_filename, obj_name, PERSIST=`false`)	write data, set PERSIST flag if want data to remain after job finishes
get(jobid, dst_filename, obj_name, DELETE=`false`)	read data, set DELETE `true` if data is only read once

Table 4.2: Main control, metadata, and storage functions exposed by Pocket's client API.

to a single metadata server, since a bucket is only managed by one metadata server, to simplify consistency management. However, a job is not fundamentally limited to one metadata server. In general, jobs can create multiple top-level buckets which hash to different metadata servers. In §4.6.2 we show that a single metadata server in our deployment supports 175K requests per second, which for the applications we study is sufficient to support jobs with thousands of lambdas.

Storage functions: Clients `put` and `get` data to/from objects at a byte granularity. Clients provide their job identifier for all operations. `Put` and `get` operations first involve a metadata lookup. Pocket enhances the basic `put` and `get` object store API calls by accepting an optional data lifetime management hint. Since ephemeral data is usually only valuable during the execution of a job, Pockets default coarse-grained behavior is to delete a job's data when the job deregisters. However, applications can set flags to override the default deletion policy for particular objects.

If a client issues a `put` with the PERSIST flag set to true, the object will persist after the job completes. The object is stored on long-running Pocket storage nodes (see §4.4.2) and will remain in Pocket until it is explicitly deleted or a (configurable) timeout period has elapsed. The ability to persist objects beyond the duration of a job is useful for piping data between jobs. If a client issues a `get` with the DELETE flag set to true, the object will be deleted as soon as it is read, allowing for more efficient garbage collection. Our analysis of ephemeral I/O characteristics for serverless analytics applications reveals that ephemeral data is often written and read only once. For example, a mapper writes an intermediate object destined to a particular reducer. Such data can be deleted as soon as it is consumed instead of waiting for the job to complete and deregister.

4.3.3 Life of a Pocket Application

We now walk through the life of a serverless analytics application using Pocket. Before launching lambdas, the application first registers with the controller and optionally provides hints about the job's characteristics (step i in Figure 4.4). The controller determines the storage tier to use (DRAM, Flash, disk) and the number of storage servers across which to distribute the job's data to meet its throughput and capacity requirements. The controller generates a weight map, described in §4.4.1, to specify the job's data placement policy and sends this information to the metadata server which it assigned for managing the job's metadata and steering client I/O requests (step ii). If the controller needs to launch new storage servers to satisfy a job's resource allocation, the job registration call stalls until these nodes are available.

When registration is complete, the job launches lambdas. Lambdas first connect to their assigned metadata server, whose IP address is provided by the controller upon job registration. Lambda clients write data by first contacting the metadata server to get the IP address and block address of the storage server to write data to. For writes to large objects which span multiple blocks, the client requests capacity allocation from the metadata server in a streaming fashion; when the capacity of a single block is exhausted, the client issues a new capacity allocation request to the metadata server. Pocket's client library internally overlaps metadata RPCs for the next block while writing data for the current block to avoid stalls. Similarly, lambdas read data by first contacting the metadata server in a similar fashion. Clients cache metadata in case they need to read an object multiple times.

When the last lambda in a job finishes, the job deregisters the job to free up Pocket resources (step iii). Meanwhile, as jobs execute, the controller continuously monitors resource utilization in storage and metadata servers (the horizontal bars on storage servers in Figure 4.4) to add/remove servers as needed to minimize cost while providing high performance (see §4.4.2).

4.3.4 Handling Node Failures

Though Pocket is not designed to provide high data durability, the system has mechanisms in place to deal with node failures. Storage servers send heartbeats to the controller and metadata servers. When a storage server fails to send heartbeats, metadata servers automatically mark its blocks as invalid. As a result, client read operations to data that was stored on the faulty storage server will return a 'data unavailable' error. Pocket currently expects the application framework to re-launch serverless tasks to regenerate lost ephemeral data. A common approach is for application frameworks to track data lineage, which is the sequence of tasks that produces each object [217, 96]. For metadata fault tolerance, Pocket supports logging of all metadata RPC operations on shared storage. When a metadata server fails, its state can be reconstructed by replaying the shared log. Controller fault tolerance can be achieved through master-slave replication, though we do not evaluate this in our study.

Hint	Impact on throughput T	Impact on capacity C	Impact on storage media
No hint (default policy)	$T = T_{\text{default}}$ (T= 50 × 8 Gb/s)	$C = C_{\text{default}}$ (C= 50 × 1960 GB)	Fill storage tiers in order of high to low performance (DRAM first, then Flash)
Latency sensitivity	-	-	If latency sensitive, use default policy above. Otherwise, choose the storage tier with the lowest cost for the estimated throughput T and capacity C required for the job.
Maximum number of concurrent lambdas N	$T = N\times$ per-λ network limit ($T = N \times 0.6$ Gb/s)	$C \propto N\times$ per-λ network limit ($C = \frac{N\times 0.6}{8\text{ Gb/s}}\times$ 1960GB)	
Total ephemeral data capacity D	$T \propto D$ ($T = \frac{D}{1960\text{GB}} \times 8$ Gb/s)	$C = D$	
Peak aggregate bandwidth B	$T = B$	$C \propto B$ ($C = \frac{B}{8\text{ Gb/s}} \times$ 1960GB)	

Table 4.3: The impact that hints provided about the application have on Pocket's resource allocation decisions for throughput, capacity and the choice of storage media (with specific examples in parentheses for our AWS deployment with `i3.2xl` instances, each with 8 cores, 60 GB DRAM, 1.9 TB Flash and ∼8 Gb/s network bandwidth).

4.4 Rightsizing Resource Allocations

Pocket's control plane elastically and automatically rightsizes cluster resources. When a job registers, Pocket's controller leverages optional hints passed through the API to conservatively estimate the job's latency, throughput and capacity requirements and find a cost-effective resource assignment, as described in §4.4.1.

In addition to rightsizing resource allocations for jobs upfront, Pocket continuously monitors the cluster's overall utilization and decides when and how to scale storage and metadata nodes based on load. §4.4.2 describes Pocket's resource scaling mechanisms and data steering policy.

4.4.1 Rightsizing Application Allocation

When a job registers, the controller first determines its *resource allocation* across three dimensions: throughput, capacity, and the choice of storage media. The controller then uses an online bin-packing algorithm to translate the resource allocation into a *resource assignment* on nodes.

Determining job I/O requirements: Pocket uses heuristics that adapt to optional hints passed through the `register_job` API. Table 4.3 lists the hints that Pocket supports and their impact on the throughput, capacity, and choice of storage media allocated for a job, with examples (in parentheses) for our deployment on AWS.

Given no hints about a job, Pocket uses a default resource allocation that conservatively overprovisions resources to achieve high performance, at high cost. In our AWS deployment, this consists of 50 `i3.2xl` nodes, providing DRAM and NVMe Flash storage with 50 GB/s aggregate throughput.

By default, Pocket conservatively assumes that a job is latency sensitive. Hence, Pocket fills the job's DRAM resources before spilling to other storage tiers, in order of increasing storage latency. If a job hints that it is not sensitive to latency, the controller does not allocate DRAM for the job and instead uses the most cost-effective storage technology for the throughput and capacity the controller estimates the job needs.

Knowing a job's maximum number of concurrent lambdas, N, allows Pocket to compute a less conservative estimate of the job's throughput requirement. If this hint is provided, Pocket allocates throughput equal to N times the peak network bandwidth limit per lambda (e.g., ∼600 Mb/s per lambda on AWS). N can be limited by the job's inherent parallelism or the cloud provider's task invocation limit (e.g., 1000 default on AWS).

Pocket's API also accepts hints for the aggregate throughput and capacity requirements of a job, which override Pocket's heuristic estimates. This information can come from profiling. When Pocket receives a throughput hint with no capacity hint, the controller allocates capacity proportional to the job's throughput allocation. The proportion is set by the storage throughput to capacity ratio on the VMs used (e.g., `i3.2xl` instances in AWS provide 1.9 TB of capacity per ∼ 8 Gb/s of network bandwidth). Vice versa, if only a capacity hint is provided, Pocket allocates throughput based on the VM capacity:throughput ratio. In the future, we plan to allow jobs to specify their average *per-lambda* throughput and capacity requirements, as these can be more meaningful than aggregate throughput and capacity hints for a job when the number of lambdas used is subject to change.

The hints in Table 4.3 can be specified by application developers or provided by the application framework. For example, the framework we use to run lambda-distributed software compilation automatically infers and synthesizes a job's dependency graph [77]. Hence, this framework can provide Pocket with hints about the job's maximum degree of parallelism, for instance.

Assigning resources: Pocket translates a job's resource allocation into a resource assignment on specific storage servers by generating a *weight map* for the job. The weight map is an associative array mapping each storage server (identified by its IP address and port) to a weight from 0 to 1, which represents the fraction of a job's dataset to place on that storage server. If a storage server is assigned a weight of 1 in a job's weight map, it will store all of the job's data. The controller sends the weight map to metadata servers, which enforce the data placement policy by routing client requests to storage servers using weighted random selection based on the weights in the job's weight map. The weight map depends on the job's resource requirements and the available cluster resources. Pocket uses an online bin-packing algorithm which first tries to fit a job's throughput, capacity and storage media allocation on active storage servers and only launches new servers if the job's requirements cannot be satisfied by sharing resources with other jobs [180]. If a job requires more resources than are currently available, the controller launches the necessary storage nodes while the application waits for its job registration command to return. Nodes take a few seconds or minutes to launch, depending on whether a new VM is required (§4.6.2).

4.4.2 Rightsizing the Storage Cluster

In addition to rightsizing the storage allocation for each job, Pocket dynamically scales cluster resources to accommodate elastic application load for multiple jobs over time. At its core, the Pocket cluster consists of a few long-running nodes used to run the controller, the minimum number of metadata servers (one in our deployment), and the minimum number of storage servers (two in our deployment). In particular, data written with the PERSIST flag described in §4.3.2, which has longer lifetime, is always stored on long-running storage servers in the cluster. Beyond these persistent resources, Pocket scales resources on demand based on load. We first describe the mechanism for horizontal and vertical scaling and then discuss the policy Pocket uses to balance cluster load by carefully steering requests across servers.

Mechanisms: The controller monitors cluster resource utilization by processing heartbeats from storage and metadata servers containing their CPU, network, and storage media capacity usage. Nodes send statistics to the controller every second. The interval is configurable.

When launching a new storage server, the controller provides the IP addresses of all metadata servers that the storage server must establish connections with to join the cluster. The new storage server registers a portion of its capacity with each of these metadata servers. Metadata servers independently manage their assigned capacity and do not communicate with each other. Storage servers periodically sends heartbeats to metadata servers.

To remove a storage server, the controller blacklists the storage server by assigning it a zero weight in the weight maps of incoming jobs. This ensures that metadata servers do not steer data from new jobs to this node. The controller instructs a randomly selected metadata server to set a 'kill' flag in the heartbeat responses of the blacklisted storage server. The blacklisted storage server waits until its capacity is entirely freed, as jobs terminate and their ephemeral data are garbage collected. The storage server then terminates and releases its resources.

When the controller launches a new metadata server, the metadata server waits for new storage servers to also be launched and register their capacity. To remove a metadata server, the controller sends a 'kill' RPC to the node. The metadata server waits for all the capacity it manages to be freed, then notifies all connected storage servers to close their connections. When all connections close, the metadata server terminates. Storage servers then register their capacity that was managed by the old metadata server across new metadata servers.

In addition to horizontal scaling, the controller manages vertical scaling. When the controller observes that CPU utilization is high and additional cores are available on a node, the controller instructs the node via a heartbeat response to use additional CPU cores.

Cluster sizing policy: Pocket elastically scales the cluster using a policy that aims to maintain overall utilization for each resource type (CPU, network bandwidth, and the capacity of each storage tier) within a target range. The target utilization range can be configured separately for each resource type and managed separately for metadata servers, long-running storage servers (which store data

written with the PERSIST flag set) and regular storage servers. For our deployment, we use a lower utilization threshold of 60% and a upper utilization threshold of 80% for all resource dimensions, for both the metadata and storage nodes. The range is empirically tuned and depends on the time it takes to add/remove nodes. Pocket's controller scales down the cluster by removing a storage server if overall CPU, network bandwidth *and* capacity utilization is below the lower limit of the target range. In this case, Pocket removes a storage server belonging to the tier with lowest capacity utilization. Pocket adds a storage server if overall CPU, network bandwidth *or* capacity utilization is above the upper limit of the target range. To respond to CPU load spikes or lulls, Pocket first tries to vertically scale CPU resources on metadata and storage servers before horizontally scaling the number of nodes.

Balancing load with data steering: To balance load while dynamically sizing the cluster, Pocket leverages the short-lived nature of ephemeral data and serverless jobs. As ephemeral data objects only live for tens to hundreds of seconds (see Figure 4.3), migrating this data to re-distribute load when nodes are added or removed has high overhead. Instead, Pocket focuses on steering data for incoming jobs across active and new storage servers joining the cluster. Pocket controls data steering by assigning specific weights for storage servers in each job's weight map. To balance load, the controller assigns higher weights to under-utilized storage servers.

The controller uses a similar approach, at a coarser granularity, to balance load across metadata servers. As noted in §4.3.2, the controller currently assigns each job to one metadata server. The controller estimates the load a job will impose on a metadata server based on its throughput and capacity allocation. Combining this estimate with metadata server resource utilization statistics, the controller selects a metadata server to use for an incoming job such that the predicted metadata server resource utilization remains within the target range.

4.5 Implementation

Pocket consists of a centralized controller, metadata servers, storage servers, and a client library. Our implementation runs on commodity cloud hardware and leverages several open source code bases.

Controller: Pocket's controller, implemented in Python, leverages the Kubernetes container orchestration system to launch and tear down metadata and storage servers, running in separate Docker containers [9]. The controller uses Kubernetes Operations (kops) to spin up and down virtual machines that run containers [8]. As explained in §4.4.2, Pocket rightsizes cluster resources to maintain a target utilization range. We implement a resource monitoring daemon in Python which runs on each node, sending CPU and network utilization statistics to the controller every second. Metadata servers also send storage tier capacity utilization statistics. We empirically tune the target utilization range based on node startup time. For example, we use a conservative target

utilization range when the controller needs to launch new VMs compared to when VMs are running and the controller simply launches containers.

Metadata management: We implement Pocket's metadata and storage server architecture on top of the Apache Crail distributed data store [4, 197]. Crail is designed for low latency, high throughput storage of arbitrarily sized data with low durability requirements. Crail provides a unified namespace across a set of heterogeneous storage resources distributed in a cluster. Its modular architecture separates the data and metadata plane and supports pluggable storage tier and RPC library implementations. While Crail is originally designed for RDMA networks, we implement a TCP-based RPC library for Pocket since RDMA is not readily available in public clouds. Like Crail, Pocket's metadata servers are implemented in Java. Each metadata server logs its metadata operations to a file on a shared NFS mount point, such that the log can be accessed and replayed by a new metadata server in case a metadata server fails.

Storage tiers: We implement three different storage tiers for Pocket. The first is a DRAM tier implemented in Java, using NIO APIs to efficiently serve requests from clients over TCP connections. The second tier uses NVMe Flash storage. We implement Pocket's NVMe storage servers on top of ReFlex, the system we described in Chapter 3, which allows clients (i.e., lambdas) to access Flash over commodity Ethernet networks with high performance. The third tier we implement is a generic block storage tier that allows Pocket to use any block storage device (e.g., HDD or SATA/SAS SSD) via a standard kernel device driver. Similar to ReFlex, this tier is implemented in C and uses DPDK for efficient, userspace networking. However, instead of using SPDK to access NVMe Flash devices from userspace, this tier uses the Linux libaio library to submit asynchronous block storage requests to a kernel block device driver. Leveraging userspace APIs for the Pocket NVMe and generic block device tiers allows us to increase performance and resource efficiency. For example, ReFlex can process up to 11× more requests per core than a conventional Linux network-storage stack [122].

Client library: Since the serverless applications we use are written in Python, we implement Pocket's application interface (Table 4.2) as a Python client library. The core of the library is implemented in C++ to optimize performance. We use `Boost` to wrap the code into a Python library. The library internally manages TCP connections with metadata and storage servers.

4.6 Evaluation

We evaluate Pocket to show it meets the performance, elasticity and cost-efficiency requirements outlined in Section 4.2.1. We show that Pocket's data plane delivers sub-millisecond I/O latencies and scalable bandwidth (§4.6.2), the control plane dynamically rightsizes cluster resources based on elastic application load (§4.6.3), and Pocket achieves the same performance as Redis while reducing cost by almost 60% for the various serverless analytics applications we study (§4.6.4).

Pocket server	EC2 server	DRAM (GB)	Storage (TB)	Network (Gb/s)	$/hr
Controller	`m5.xl`	16	0	∼8	0.192
Metadata	`m5.xl`	16	0	∼8	0.192
DRAM	`r4.2xl`	61	0	∼8	0.532
NVMe	`i3.2xl`	61	1.9	∼8	0.624
SSD	`i2.2xl`	61	1.6	$\lesssim 2$	1.705[2]
HDD	`h1.2xl`	32	2	∼8	0.468

Table 4.4: Type and cost of Amazon EC2 VMs used for Pocket

4.6.1 Methodology

We deploy Pocket on Amazon Web Service (AWS). We use EC2 instances to run Pocket storage, metadata, and controller nodes. We use four different kinds of storage media: DRAM, NVMe-based Flash, SATA/SAS-based Flash (which we refer to as SSD), and HDD. DRAM servers run on `r4.2xl` instances, NVMe Flash servers run on `i3.2xl` instances, SSD servers run on `i2.2xl` instances, and HDD servers run on `h1.2xl` instances. We choose the instance families based on their local storage media, shown in Table 4.4. We choose the VM size to provide a good balance of network bandwidth and storage capacity for the serverless applications we study.

We run Pocket storage and metadata servers as containers on EC2 VMs, orchestrated with Kubernetes v1.9. We use AWS Lambda as our serverless computing platform. We enable lambdas to access Pocket EC2 nodes by deploying them in the same virtual private cloud (VPC). We configure lambdas with 3 GB of memory. Amazon allocates lambda compute resources proportional to memory resources [24]. We compare Pocket's performance and cost-efficiency to ElastiCache Redis (cluster-mode enabled) and Amazon S3 [19, 128, 22]. We conducted experiments in April 2018.

We study three different serverless analytics applications, described below. The applications differ in their degree of parallelism, ephemeral object size distribution (Figure 4.2), and throughput requirements.

Video analytics: We use the Thousand Island Scanner (THIS) for distributed video processing [167]. Lambdas in the first stage read compressed video frame batches, decode, and write the decoded frames to ephemeral storage. Each lambda fetches a 250 MB decoder executable from S3 as it does not fit in the AWS Lambda deployment package. Each first stage lambda then launches second stage lambdas, which read decoded frames from ephemeral storage, compute a MXNET deep learning classification algorithm and output an object detection result. We use a 25 minute video with 40K 1080p frames. We tune the batch size for each stage to minimize the job's end-to-end execution time; the first stage consists of 160 lambdas while the second has 305 lambdas.

MapReduce Sort: We implement a MapReduce sort application on AWS Lambda, similar to PyWren [116]. Map lambdas fetch input files from long-term storage (we use S3) and write

[2]The cost of the `i2` instance is particularly high since it is an old generation instance that is being phased out by AWS and replaced by the newer generation `i3` instances with NVMe Flash devices.

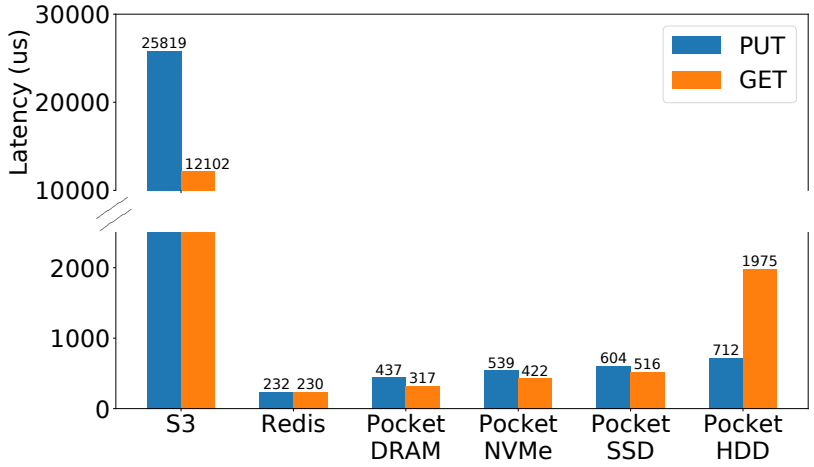


Figure 4.5: Unloaded latency for 1KB I/Os from lambda clients using different storage systems.

intermediate files to ephemeral storage. Reduce lambdas merge and sort intermediate data and upload output files to long-term storage. We run a 100 GB sort, which generates 100 GB of ephemeral data. We run the job with 250, 500, and 1000 lambdas.

Distributed software compilation (λ-cc): We use `gg` to infer software build dependency trees and invoke lambdas to compile source code with high parallelism [6, 77]. Each lambda fetches its dependencies from ephemeral storage, computes (i.e., compiles, archives or links), and writes its output to ephemeral storage, including the final executable for the user to download. We present results for compiling the `cmake` project source code. This build job has a maximum inherent parallelism of 650 tasks and generates a total of 850 MB ephemeral data. Object size ranges from 10s of bytes to MBs, as shown in Figure 4.2.

4.6.2 Microbenchmarks

Storage request latency: Figure 4.5 compares the 1 KB request latency of S3, Redis, and various Pocket storage tiers measured from a lambda client. Pocket-DRAM, Pocket-NVMe and Redis latency is below 540 μs, which is over 45$\times$ faster than S3. The latency of the Pocket-SSD and Pocket-HDD tiers is higher due to higher storage media access times. Pocket-HDD `get` latency is higher than `put` latency since lambdas issue random reads while writes are sequential; the metadata server routes writes to sequential logical block addresses. Pocket-DRAM has higher latency than Redis mainly due to the metadata lookup RPC, which takes 140 μs. While Redis cluster clients simply hash keys to Redis nodes, Pocket clients must contact a metadata server. While this extra RPC increases request latency, it allows Pocket to optimize data placement per job and dynamically scale the cluster without redistributing data across nodes.

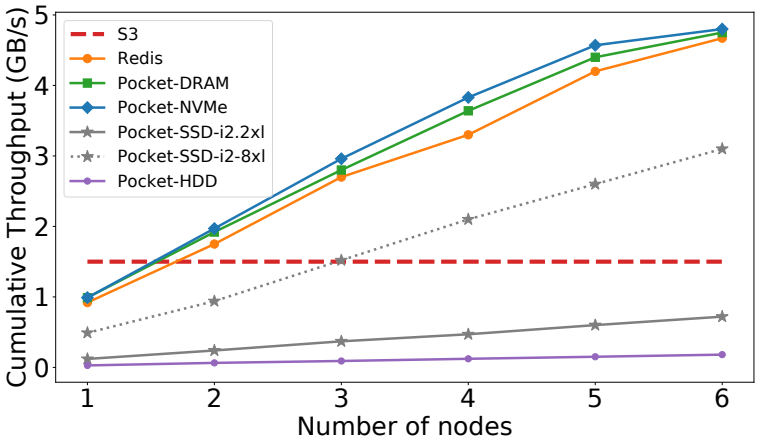


Figure 4.6: Total GB/s for 1MB requests from 100 lambda clients using different storage systems.

Storage request throughput: We measure the `get` throughput of S3, Redis (`cache.r4.2xl`) and various Pocket storage tiers by issuing 1 MB requests from 100 concurrent lambdas. In Figure 4.6, we sweep the number of nodes in the Redis and Pocket clusters and compare the cumulative throughput to that achieved with S3. Pocket-DRAM, Pocket-NVMe and Redis all achieve similar throughput. With a single node, the bottleneck is the 1 GB/s VM network bandwidth. With two nodes, Pocket's DRAM and NVMe tiers achieve higher throughput than S3. Pocket's SSD and HDD tiers have significantly lower throughput. The HDD tier is limited by the 40 MB/s random access bandwidth of the disk on each node. The SSD tier is limited by poor networking (less than ∼2 Gb/s) on the old generation `i2.2xl` instances. Hence, we also plot the throughput using `i2.8xl` instances which have 10 Gb/s networking. The bottleneck becomes the 500 MB/s throughput limit of the SATA/SAS SSD.

We focus the rest of our evaluation of Pocket on the DRAM and NVMe Flash tiers as they demand the highest data plane software efficiency due to the technology's low latency and high throughput. We also find that in our AWS deployment, the DRAM and NVMe tiers offer significantly higher performance-cost efficiency compared to the HDD and SSD tiers. For example, NVMe Flash servers, which run on on `i3.2xl` instances, provide 1 GB/s per 1900 GB capacity at a cost of $0.624/hour. Meanwhile, HDD servers, on `h1.2xl` instances, provide only 40 MB/s per 2000 GB capacity at a cost of $0.468/hour. Thus, the NVMe tier offers 19.7× higher throughput per GB per dollar.

Metadata throughput: We measure the number of metadata operations that a metadata server can handle per second. A single core metadata server on the `m5.xl` instance supports up to 90K operations per second and up to 175K operations per second with four cores. The peak metadata request rate we observe for the serverless analytics applications we study is 75 operations per second per lambda. Hence, a multi-core metadata server can support jobs with thousands of lambdas.

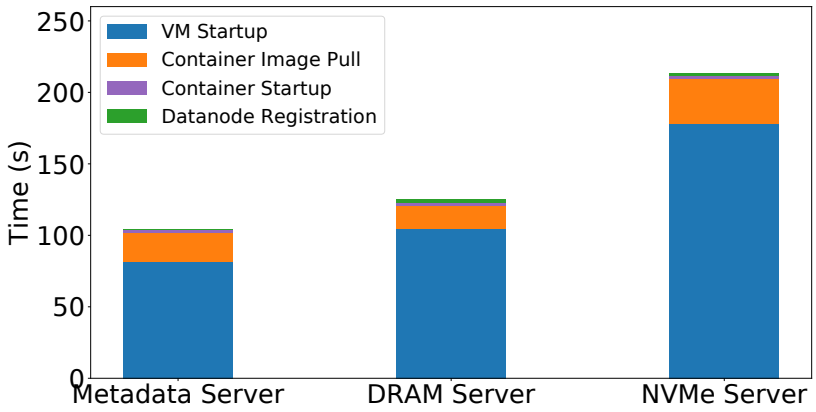


Figure 4.7: Pocket node startup time breakdown.

Adding/removing servers: Since Pocket runs in containers on EC2 nodes, we measure the time it takes to launch a VM, pull the container image, and launch the container. Pocket storage servers must also register their storage capacity with metadata servers to join the cluster. Figure 4.7 shows the time breakdown. VM launch time varies across EC2 instance types. The container image for the metadata server and DRAM server has a compressed size of 249 MB while the Pocket-NVMe compressed container image is 540 MB due to dependencies for DPDK and SPDK to run ReFlex. The image pull time depends on the VM's network bandwidth. The VM launch time and container image pull time only need to be done once when the VM is first started. Once the VM is warm, meaning the image is available locally, starting and stopping containers takes only a few seconds. The time to terminate a VM is tens of seconds.

4.6.3 Rightsizing Resource Allocations

We now evaluate Pocket with the three different serverless applications described in §3.4.1.

Rightsizing with application hints: Figure 4.8 shows how Pocket leverages user hints to make cost-effective resource allocations, assuming each hint is provided in addition to the previous ones. With no knowledge of application requirements, Pocket defaults to a policy that spreads data for a job across a default allocation of 50 nodes, filling DRAM first, then Flash. With knowledge of the maximum number of concurrent lambdas (250, 160, and 650 for the sort, video analytics and λ-cc jobs, respectively), Pocket allocates lower aggregate throughput than the default allocation while maintaining similar job execution time (within 4% of the execution time achieved with the default allocation). Furthermore, these jobs are not sensitive to latency; the sort job and the first stage of the video analytics job are throughput intensive while λ-cc and the second stage of the video analytics job are compute limited. The orange bars in Figure 4.8 show the cost savings of using NVMe Flash as opposed to DRAM when the latency insensitivity hint is provided for these jobs.

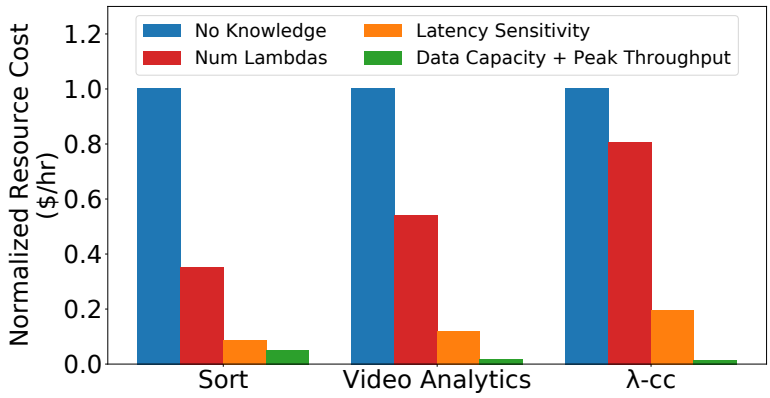


Figure 4.8: Resource cost savings achieved by Pocket when hints are provided cumulatively.

The green bar shows the relative resource allocation cost when applications provide explicit hints for their capacity and peak throughput requirements; such hints can be obtained from a profiling run. Across all scenarios, each job's execution time remains within 4% of its execution time with the default resource allocation.

Reclaiming capacity using hints: Figure 4.9 shows the capacity used over time for the video analytics job, with and without data lifetime management hints. All ephemeral data in this application is written and read only once, since each first stage lambda writes ephemeral data destined to a single second stage lambda. Hence for all `get` operations, this job can make use of the DELETE hint which informs Pocket to promptly garbage collect an object as soon as it has been read. By default, when the DELETE hint is not specified, Pocket waits until the job deregisters to delete the job's data. The job in Figure 4.9 completes at the 158 second mark. We show that leveraging the DELETE hint allows Pocket to reclaim capacity more promptly, making more efficient use of resources as this capacity can be offered to other jobs.

Rightsizing cluster size: Elastic and automatic resource scaling is a key property of Pocket. Figure 4.10 shows how Pocket scales cluster resources as multiple jobs register and deregister with the controller. Job registration and deregistration times are indicated by upwards and downwards arrows along the x-axis, respectively. In this experiment, we assume Pocket receives capacity and throughput hints for each job's requirements. The first job is a 10 GB sort application requesting 3 GB/s, the second job is a video analytics application requesting 2.5 GB/s and the third job is a different invocation of a 10 GB sort also requesting 3 GB/s. Each storage server provides 1 GB/s. We use a minimum of two storage servers in the cluster. We provision seven VMs for this experiment and ensure that storage server containers are locally available, such that when the controller launches new storage servers, only container startup and capacity registration time is included.

Figure 4.10 shows that Pocket quickly and effectively scales the allocated storage bandwidth (dotted line) to meet application throughput demands (solid line). The spike surpassing the allocated

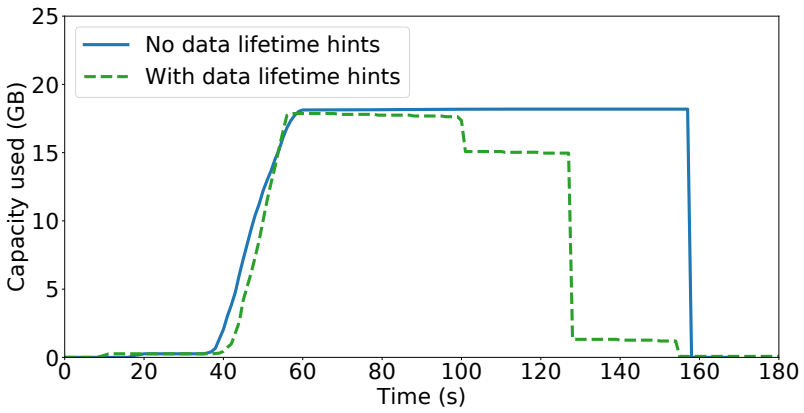


Figure 4.9: Capacity usage benefits with garbage collection hints in Pocket. With the DELETE hint for `get` operations in a video analytics job, Pocket can readily reclaim capacity by deleting objects after they have been read versus waiting for the job to complete.

throughput is due to a short burst in EC2 VM network bandwidth. The VMs provide 'up to 10 Gb/s', but since we typically observe a ∼8 Gb/s bandwidth limit in practice, the controller allocates throughput assuming each node provides 8 Gb/s. As the controller rightsizes resources for each job, job execution time stays within 5% of its execution time when running on 50 nodes, the conservative default resource allocation. If the controller had to spin up new VMs to accommodate a job's requirements instead of just launching containers, the job's start time would be delayed by up to 215 seconds (see EC2 NVMe server startup time in Figure 4.7) since the `register_job` call blocks until the required storage servers are available.

4.6.4 Comparison to S3 and Redis

Job execution time: Figure 4.11 plots the per-lambda execution time breakdown for the MapReduce 100 GB sort job, run with 250, 500, and 1000 concurrent lambdas. The purple bars show the time spent fetching original input data and writing final output data to S3 while the blue bars compare the time for ephemeral data I/O with S3, Redis and Pocket-NVMe. S3 does not support sufficient request rates when the job is run with 500 or more lambdas. S3 returns errors, advising to reduce the I/O rate. Pocket provides similar throughput to Redis, however since the application is not sensitive to latency, Pocket uses NVMe Flash instead of DRAM to reduce cost.

Similarly, for the video analytics job, we observe that Pocket-NVMe achieves the same performance as Redis. However, using S3 for the video analytics job increases the average time spent on ephemeral I/O by each lambda in the first stage (video decoding) by 3.2× and 4.1× for lambdas in the second stage (MXNET classification), compared to using Pocket or Redis.

The performance of the distributed compilation job (λ-cc cmake) is limited by lambda CPU resources [124]. A software build job has inherently limited parallelism; early-stage lambdas compile

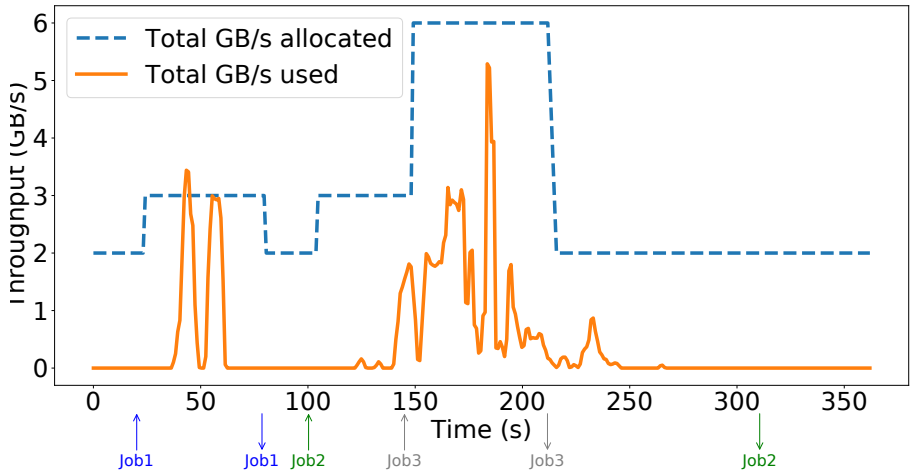


Figure 4.10: Pocket's controller dynamically scales cluster resources to meet I/O requirements as jobs enter and leave the system.

independent files in parallel, however lambdas responsible for archiving and linking are serialized as they depend on the outputs of the early-stage lambdas. We observe that the early-stage lambdas are compute-bound on current serverless infrastructure. Although using Pocket or Redis reduces the fraction of time each lambda spend on ephemeral I/O, the overall execution time for this job remains the same as when using S3 for ephemeral storage, since the bottleneck is dependencies on compute-bound lambdas.

Cost analysis: Table 4.4 shows the hourly cost of running Pocket nodes on EC2 VMs in April 2018. Our minimum size Pocket cluster, consisting of one controller node, one metadata server and two `i3.2xl` storage nodes costs $1.632 per hour on EC2. However, Pocket's fixed cost can be amortized as the system is designed to support multiple concurrent jobs from one or more tenants. We intend for Pocket to be operated by a cloud provider and offered as a storage service with a pay-what-you-use cost model for users, similar to the cost model of serverless computing platforms. Hence, for our cost analysis, we derive fine-grain resource costs, such as the cost of a CPU core and the cost of storage per GB, using AWS EC2 instance pricing. For example, we calculate NVMe Flash $/GB by taking the difference between `i3.2xl` and `r4.2xl` instance costs (since these VMs have the same CPU and DRAM configurations but `i3.2xl` includes a 1900 GB NVMe drive) and dividing by the GB capacity of the `i3.2xl` NVMe drive.

Using this fine-grain resource pricing model for Pocket, Table 4.5 compares the cost of running the 100 GB sort, video analytics and distributed compilation jobs with S3, ElastiCache Redis, and Pocket-NVMe. We use reduced redundancy pricing for S3 and assume the GB-month cost is charged hourly [21]. We base Redis costs on the price of entire VMs, not only the resources consumed, since ElastiCache Redis clusters are managed by individual users rather than cloud providers. Pocket

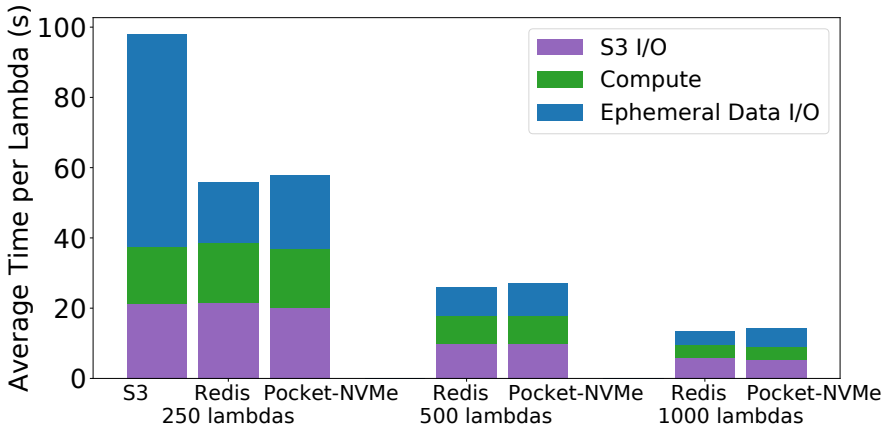


Figure 4.11: Execution time breakdown of 100GB sort.

Job	S3	Redis	Pocket
100 GB sort	0.05126	5.320	2.1648
Video analytics	0.00034	1.596	0.6483
λ-cc cmake	0.00005	1.596	0.6480

Table 4.5: Hourly ephemeral storage cost (in USD)

achieves the same performance as Redis for all three jobs while saving 59% in cost. S3 is still orders of magnitude cheaper. However, S3's cloud provider based cost is not a fair comparison to the cloud user based cost model we use for Pocket and Redis. Furthermore, while the λ-cc job has similar performance with Pocket, Redis and S3 due to a lambda compute bottleneck, the video analytics and sort job execution time is 40 to 65% higher with S3.

4.7 Discussion

Choice of API: Pocket's simple `get/put` interface provides sufficient functionality for the applications we studied. Lambdas in these jobs consume entire data objects that they read and they do not require updating or appending files. However, POSIX-like I/O semantics for appending or accessing parts of objects could benefit other applications. Pocket's `get/put` API is implemented on top of Apache Crail's append-only stream abstraction which allows clients to read at file offsets and append to files with single-writer semantics [196]. Thus, Pocket's API could easily be modified to expose Crail's I/O semantics. Other operators such as filters or multi-gets could also help optimize the number of RPCs and bytes transferred. The right choice of API for ephemeral storage remains an open question.

Security: Pocket uses access control to secure applications in a multi-tenant environment. To

prevent malicious users from accessing other tenants' data, metadata servers issue single-use certificates to clients which are verified at storage servers. An I/O request that is not accompanied with a valid certificate is denied. Clients communicate with metadata servers over SSL to protect against man in the middle attacks. Users set cloud network security rules to prevent TCP traffic snooping on connections between lambdas and storage servers. Alternatively, users can encrypt their data. Pocket does not currently prevent jobs from issuing higher load than specified in job registration hints. Request throttling can be implemented at metadata servers to mitigate interference when a job tries to exceed its allocation.

Learning job characteristics: Pocket currently relies on user or application framework hints to cost-effectively rightsize resource allocations for a job. Currently, Pocket does not autonomously learn application properties. Since users may repeatedly run jobs on different datasets, as many data analytics and modern machine learning jobs are recurring [140], Pocket's controller can maintain statistics about previous invocations of a job and use this information combined with machine learning techniques to rightsize resource allocations for future runs [123, 15]. We plan to explore this in future work.

Applicability to other cloud platforms: While we evaluate Pocket on the AWS cloud platform, the system addresses a real problem applicable across all cloud providers as no available platform provides an optimized way for serverless tasks to exchange ephemeral data. Pocket's performance will vary with network and storage capabilities of different infrastructure. For example, if a low latency network is available, the DRAM storage tier provides significantly lower latency than the NVMe tier. Such variations emphasize the need for a control plane to automate resource allocation and data placement.

Applicability to other cloud workloads: Though we presented Pocket in the context of ephemeral data sharing in serverless analytics, Pocket can also be used for other applications that require distributed, scalable temporary storage. For instance, Google's Cloud Dataflow, a fully-managed data processing service for streaming and batch data analytics pipelines, implements the shuffle operator – used for transforms such as `GroupByKey` – as part of its service backend [86]. Pocket can serve as fast, elastic storage for the intermediate data generated by shuffle operations in this kind of service.

Reducing cost with resource harvesting: Cloud jobs are commonly over-provisioned in terms CPU, DRAM, network, and storage resources due to the difficulty of rightsizing general jobs and the need to accommodate diurnal load patterns and unexpected load spikes. The result is significant capacity underutilization at the cluster level [35, 205, 68]. Recent work has shown that the plethora of allocated but temporarily unused resources provide a stable substrate that can be used to run analytics job [47, 220]. We can similarly leverage harvested resources to dramatically reduce the total cost of running Pocket. Pocket's storage servers are particularly well suited to run on temporarily idle resource as ephemeral data has short lifetime and low durability requirements.

4.8 Related Work

Section 4.2 provided an overview of related storage systems. Here we discuss work related to resource scaling and data placement policies. We also describe how Pocket fits in the context of fully managed data warehouses offered by today's cloud providers.

Elastic resource scaling: Various reactive [81], predictive [57, 127, 171, 72, 152, 202, 208] and hybrid [53, 103, 80, 150] approaches have been proposed to automatically scale resources based on demand [168, 151]. Muse takes an economic approach, allocating resources to their most efficient use based on a utility function that estimates the impact of resource allocations on job performance [51]. Pocket provisions resources upfront for a job based on hints and conservative heuristics while using a reactive approach to adjust cluster resources over time as jobs enter and leave the system.

Pocket's reactive scaling is similar to Horizontal Pod autoscaling in Kubernetes which collects multidimensional metrics and adjusts resources based on utilization ratios [7]. Petal [130] and the controller by Lim et al. [135] propose data re-balancing strategies in elastic storage clusters while Pocket avoids redistributing short-lived data due to the high overhead. CloudScale [182], Elastisizer [100], CherryPick [16], and other systems [204, 214, 123] take an application-centric view to rightsize a job at the coarse granularity of traditional VMs as opposed to determining fine-grain storage requirements. Nevertheless, the proposed cost and performance modeling approaches can also be applied to Pocket to autonomously learn job resource preferences.

Intelligent data placement: Mirador is a dynamic storage service that optimizes data placement for performance, efficiency, and safety [211]. Mirador focuses on long-running jobs (minutes to hours), while Pocket targets short-term (seconds to minutes) ephemeral storage. Tuba manages geo-replicated storage and, similar to Pocket, optimizes data placement based on performance and cost constraints received from applications [28]. Extent-based Dynamic Tiering (EDT) uses access pattern simulations and monitoring to find a cost-efficient storage solution for a workload across multiple storage tiers [90]. The access pattern of ephemeral data is often simple (e.g., write-once-read-once) and the data is short-lived, hence it is not worth migrating between tiers. Multiple systems make storage configuration recommendation based on workload traces [26, 195, 14, 149, 17]. Given I/O traces for a job, Pocket could apply similar techniques to assign resources when a job registers.

Fully managed data warehousing: Cloud providers offer fully managed infrastructure for querying large amounts of structured data with high parallelism and elasticity. Examples include Amazon Redshift [20], Google BigQuery [88], Azure SQL Data Warehouse [147], and Snowflake [58]. These systems are designed to support relational queries and high data durability, while Pocket is designed for elastic, fast, and fully managed storage of data with low durability requirements. However, a cloud data warehouse like Snowflake, which currently stores temporary data generated by query operators on local disk or S3, could leverage Pocket to improve elasticity and resource utilization.

4.9 Conclusion

General-purpose analytics on serverless infrastructure presents unique opportunities and challenges for performance, elasticity and resource efficiency. We analyzed challenges associated with efficient data sharing and presented Pocket, an ephemeral data store for serverless analytics. In a similar spirit to serverless computing, Pocket aims to provide a highly elastic, cost-effective, and fine-grained storage solution for analytics workloads. Pocket achieves these goals using a strict separation of responsibilities for control, metadata, and data management. To the best of our knowledge, Pocket is the first system designed specifically for ephemeral data sharing in serverless analytics workloads. Our evaluation on AWS demonstrates that Pocket offers high performance data access for arbitrary size data sets, combined with automatic fine-grain scaling, self management and cost effective data placement across multiple storage tiers.

Chapter 5

Selecta: Automatic Cloud Storage Configuration for Data Analytics

5.1 Introduction

As discussed in Section 2.2.2, determining the right cloud configuration for analytics applications is challenging, yet critical for both performance and cost. Even if we limit ourselves to a single instance type and focus on optimizing performance, the choice of storage configuration for a particular application remains non-trivial. Figure 2.2, discussed in Section 2.2.2, showed an example. Part of the challenge is that analytics workloads access multiple data streams, including input and output files, logs, and intermediate data (e.g., shuffle and broadcast). Each data stream has distinct characteristics in terms of access frequency, access patterns, and data lifetime, which make different streams more suitable for different types of storage devices.

We present *Selecta*, a tool that learns near-optimal VM and storage configurations for analytics applications for user-specified performance-cost objectives. Selecta targets analytics jobs that are frequently or periodically re-run on newly arriving data [12, 76, 163]. A configuration is defined by the type of cloud instance (core count and memory capacity) along with the storage type and capacity used for input/output data and for intermediate data. To predict application performance for different configurations, Selecta applies latent-factor collaborative filtering, a machine-learning technique commonly used in recommender systems [39, 172, 40, 67, 68]. Selecta uses sparse performance data for training applications profiled on various cloud configurations, as well as performance measurements for the target application profiled on only two configurations. Selecta leverages the sparse training data to learn significantly faster and more cost-effectively than exhaustive search. The approach also improves on recent systems such as CherryPick and Ernest whose performance

Block Storage	Seq Read MB/s	Seq Write MB/s	Rand Read IOPS	Rand Write IOPS	Rand Rd/Wr IOPS
r-HDD	135	135	132	132	132
r-SSD	165	165	3,068	3,068	3,068
l-NVMe	490	196	103,400	35,175	70,088

Table 5.1: Performance for 500 GB volumes. Sequential I/Os are 128 KB, random I/Os are 4 KB.

prediction models require more information about the target application and hence require more application runs to converge [15, 204]. Moreover, past work does not consider the heterogeneous cloud storage options or the varying preferences of different data streams within each application [214].

We evaluate Selecta with over one hundred Spark SQL and ML workloads, each with two different dataset scaling factors. We show that Selecta chooses a near-optimal performance configuration (within 10% of optimal) with 94% probability and a near-optimal cost configuration with 80% probability. We also analyze Selecta's sensitivity to various parameters such as the amount of information available for training workloads or the target application.

5.2 Motivation and Background

We discuss current approaches for selecting a cloud storage configuration and elaborate on the challenges involved, which we briefly introduced in Section 2.3.2. Throughout, we use the notation *l*- and *r*- to refer to local and remote storage options, respectively. For example, *l*-NVMe refers to local NVMe Flash.

5.2.1 Current Approaches

Conventional configurations: Input/output files for data analytics jobs are traditionally stored in a distributed file system, such as HDFS or object storage systems such as Amazon S3 [187, 22]. Intermediate data is typically read/written to/from a dedicated local block storage volume on each node (i.e., *l*-SSD or *l*-NVMe) and spilled to *r*-HDD if extra capacity is needed. In typical Spark-as-a-service cloud deployments, two remote storage volumes are provisioned by default per instance: one for the instance root volume and one for logs [61].

Existing tools: Recent work focuses on automatically selecting an optimal VM configuration in the cloud [214, 204, 15]. However, these tools tend to ignore the heterogeneity of cloud storage options, at best distinguishing between 'fast' and 'slow'. In the next section, we discuss the extent of the storage configuration space.

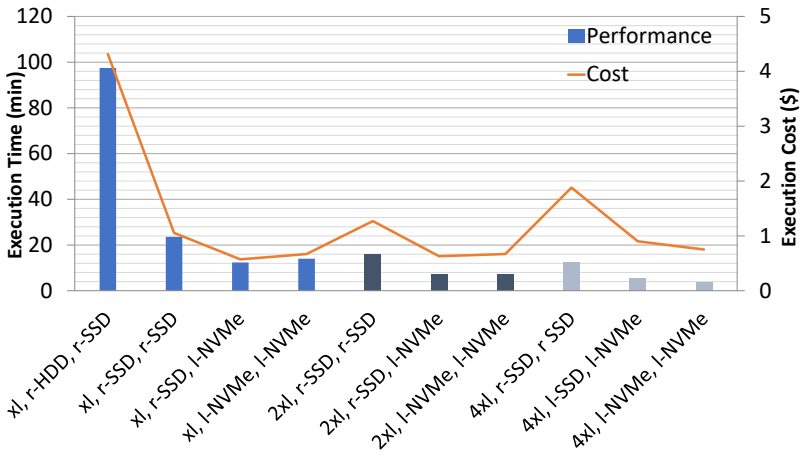


Figure 5.1: Comparison of execution time and cost for TPC-DS query 64 on various VM and storage configurations, defined as <VM size, storage for input/output data, storage for intermediate data>.

5.2.2 Challenges

Complex configuration space: Cloud storage comes in multiple flavors: object storage (e.g., Amazon S3 [22]), file storage (e.g., Azure Files [144]), and block storage (e.g., Google Compute Engine Persistent Disks [85]). Block and object storage are most commonly used for data analytics. Block storage is further sub-divided into hardware options: cold or throughput-optimized hard drive disk, SAS SSD, or NVMe Flash. Block storage can be local (directly attached) or remote (over the network) to an instance. Local block storage is ephemeral; data persists only as long as the instance is running. Remote volumes persist until explicitly deleted by the user.

Table 5.1 compares three block storage options available in Amazon Web Services (AWS). Each storage option provides a different performance, cost, and flexibility trade-off. For instance, *l*-NVMe storage offers the highest throughput and lowest latency at higher cost per bit. Currently, cloud providers typically offer NVMe in fixed capacity units directly attached to select instance types, charged per second or hour. AWS currently charges $0.023 more per hour for an instance with 475 GB of NVMe Flash compared to without NVMe. In contrast, S3 fees are based on capacity ($0.023 per GB/month) and bandwidth ($0.004 per 10K GET requests) usage.

In addition to the storage configuration, users must choose from a variety of VM types to determine the right number of CPU cores and memory, the number of VMs, and their network bandwidth. These choices often affect storage and must be considered together. For example, on instances with 1 Gb/s network bandwidth, the network limits the sequential throughput achievable with *r*-HDD and *r*-SSD storage volumes in Table 5.1.

Performance-cost objectives: While configurations with the most CPU cores, the most memory, and fastest storage generally provide the highest performance, optimizing for runtime cost is much more difficult. Systems designed to optimize a specific objective (e.g., predict the configuration

that maximizes performance or minimizes cost) are generally not sufficient to make recommendations for more complex objectives (e.g., predict the configuration that minimizes execution time within a specific budget). By predicting application execution time on candidate configurations, our approach remains general. Unless otherwise specified, we refer to cost as the cost of executing an application.

Heterogeneous application data: We classify data managed by distributed data analytics frameworks (e.g., Spark [217]) into two main categories: *input/output data* which is typically stored long-term and *intermediate data* which lives for the duration of job execution. Examples of intermediate data include shuffle data exchanged between mappers and reducers, broadcast variables, and cached dataset partitions spilled from memory. These streams typically have distinct access frequency, data lifetime, access type (random vs. sequential), and I/O size. For example, input/output data is generally long-lived and sequentially accessed, whereas intermediate data is short-lived and most accesses are random.

Storage decisions are complex: Selecting the right configuration for a job significantly reduces execution time and cost, as shown in Figure 5.1, which compares a Spark SQL query (TPC-DS query 64) on various VM and storage configurations in an 8-node cluster. We consider 3 `i3` VM instance sizes in EC2 (`xl, 2xl, and 4xl`) and heterogeneous storage options for input/output and intermediate data. The lowest performing configuration has 24× the execution time of the best performing configuration. Storing input/output data on r-SSD and intermediate data on l-NVMe (the lowest cost configuration) has 7.5× lower cost than storing input/output data on r-HDD and intermediate data on r-SSD.

5.3 Selecta Design

5.3.1 Overview

Selecta is a tool that automatically predicts the performance of a target application on a set of candidate configurations. As shown in Figure 5.2, Selecta takes as input: i) execution time for a set of training applications on several configurations, ii) execution time for the target application on two reference configurations, and iii) a performance-cost objective for the target application. A configuration is defined by the number of nodes (VM instances), the CPU cores and memory per node, as well as the storage type and capacity used for input/output data and for intermediate data. Selecta uses latent factor collaborative filtering (see §5.3.2) to predict the performance of the target application on the remaining (non-reference) candidate configurations. With these performance predictions and the per unit time cost of various VM instances and storage options, Selecta can recommend the right configuration for the user's performance-cost objective. For example, Selecta can recommend configurations that minimize execution time, minimize cost, or minimize execution time within a specific budget.

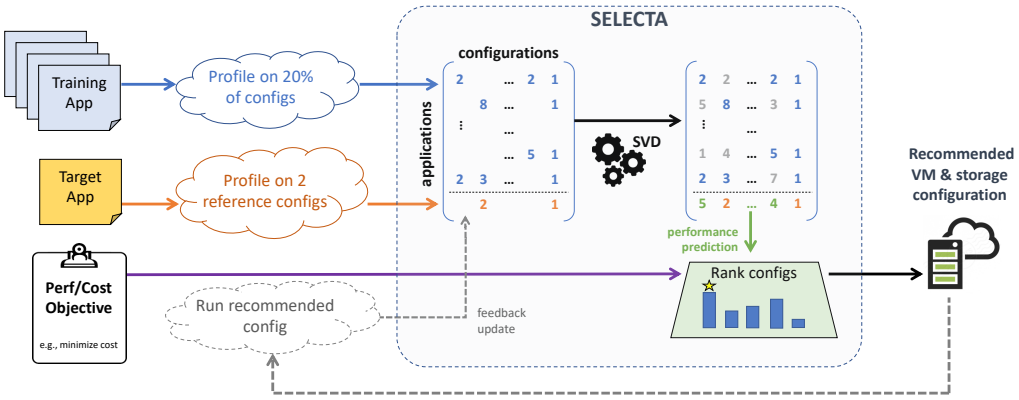


Figure 5.2: An overview of performance prediction and configuration recommendation with Selecta.

As new applications are launched over time, these performance measurements become part of Selecta's growing training set and accuracy improves (see § 5.4.4). We also feed back performance measurements after running a target application on a configuration recommended by Selecta — this helps reduce measurement noise and improve accuracy. Since Selecta takes ∼1 minute to generate a new set of predictions (the exact runtime depends on the training matrix size), a user can re-run Select when re-launching the target application with a new dataset to get a more accurate recommendation. In our experiments, the recommendations for each target application converge after two feedback iterations. The ability to grow the training set over time also provides Selecta with a mechanism for expanding the set of configurations it considers. Initially, the configuration space evaluated by Selecta is the set of configurations that appear in the original training set. When a new configuration becomes available and Selecta receives profiling data for applications on this configuration, the tool will start predicting performance for all applications on this configuration.

5.3.2 Predicting Performance

Prediction approach: Selecta uses collaborative filtering to predict the performance of a target application on candidate configurations. We choose collaborative filtering as it is agnostic to the details of the data analytics framework used (e.g., Spark vs. Storm) and it allows us to leverage *sparse* training data collected *across applications* and configurations [170]. While systems such as CherryPick [15] and Ernest [204] build performance models based solely on training data for the target application, Selecta's goal is to leverage training data available from multiple applications to converge to accurate recommendations with only two profiling runs of a target application. We discuss alternatives to collaborative filtering to explain our choice.

Content-based approaches, such as as linear regression, random forests, and neural network models, build a model from features such as application characteristics (e.g., GB of shuffle data read/written) and configuration characteristics (e.g., I/O bandwidth or the number of cores per VM).

We find that unless inputs features such as the average CPU utilization of the target application *on the target configuration* are used in the model, content-based predictors do not have enough information to learn the compute and I/O requirements of applications and achieve low accuracy. Approaches that require running target applications on all candidate configurations to collect feature data are impractical.

Another alternative is to build performance prediction models based on the structure of an analytics framework, such as the specifics of the map, shuffle, and reduce stages in Spark [110, 222]. This leads to framework-specific models and may require re-tuning or even re-modeling as framework implementations evolve (e.g., as the CPU efficiency of serialization operations improves).

Latent factor collaborative filtering: Selecta's collaborative filtering model transforms applications and configurations to a latent factor space [39]. This space characterizes applications and configurations in terms of latent (i.e., 'hidden') features. These features are *automatically inferred* from performance measurements of training applications [170]. We use a matrix factorization technique known as Singular Value Decomposition (SVD) for the latent factor model. SVD decomposes an input matrix P, with rows representing applications and columns representing configurations, into the product of three matrices, U, λ, and V. Each element p_{ij} of P represents the normalized performance of application i on configuration j. The latent features are represented by singular values in the diagonal matrix λ, ordered by decreasing magnitude. The matrix U captures the strength of the correlation between a row in P and a latent feature in λ. The matrix V captures the strength of the correlation between a column in P and a latent feature in λ. Although the model does not tell us what the latent features physically represent, a hypothetical example of a latent feature is random I/O throughput. For instance, Selecta could infer how strongly an application's performance depends on random I/O throughput and how much random I/O throughput a configuration provides.

One challenge for running SVD is the input matrix P is sparse, since we only have the performance measurements of applications on certain configurations. In particular, we only have two entries in the target application row and filling in the missing entries corresponds to predicting performance on the other candidate configurations. Since performing SVD matrix factorization requires a fully populated input matrix P, we start by randomly initializing the missing entries and then run Stochastic Gradient Descent (SGD) to update these unknown entries using an objective function that minimizes the mean squared error on the *known* entries of the matrix [45]. The intuition is that by iteratively decomposing and updating the matrix in a way that minimizes the error for known entries, the technique also updates unknown entries with accurate predictions. A useful rule of thumb is to retain enough singular values to make up 90% of the energy in matrix λ [132]. That is, the sum of the squares of the retained singular values should be at least 90% of the sum of the squares of all the singular values. Selecta uses the Python sci-kit `surprise` library for SVD [105].

5.3.3 Using Selecta

New target application: The first time an application is presented to Selecta, it is profiled on two reference configurations which, preferably, are far apart in their compute and storage resource attributes. Selecta requires that reference configurations remain fixed across all applications, since performance measurements are normalized to a reference configuration before running SVD. Profiling application performance involves running the application to completion and recording execution time and CPU utilization (including iowait) over time.

Defining performance-cost objectives: After predicting application performance across all configurations, Selecta recommends a configuration based on a user-defined ranking function. For instance, to minimize runtime cost, the ranking function is min(runtime $\times$ cost/hour). While choosing a storage technology (e.g., SSD vs. NVMe Flash), Selecta must also consider the application's storage capacity requirements. Selecta leverages statistics from profiling runs available in Spark monitoring logs to determine the intermediate (shuffle) data and and input/output data capacity [189].

Adapting to changes: Recurring jobs and their input datasets are likely to evolve. To detect changes in application characteristics that may impact the choice of optimal configuration, Selecta relies on CPU utilization information from both initial application profiling and subsequent executions rounds. When an application is first introduced to the system, Selecta assigns a unique ID to store application specific information such as iowait CPU utilization. Whenever an application is re-executed, Selecta compares the current iowait time to the stored configuration. Depending on the difference in iowait time, Selecta will either compute a refined prediction based on available measurements or treat the workload as new application, starting a new profiling run.

Dealing with noise in the cloud: An additional challenge for recommending optimal configurations is noise on public cloud platforms, which arises due to interference with other tenants, hardware heterogeneity, or other sources [179]. To account for noise, Selecta relies on the feedback of performance and CPU utilization measurements. As measurements are fed into the system, Selecta averages performance and CPU utilization and uses reservoir sampling to avoid high skew from outliers [206]. Selecta keeps a configurable number of sample points for each entry in the application-configuration matrix (e.g., three) to detect changes in applications as described above. If a particular run is heavily impacted by noise such that the compute and I/O bottlenecks differ significantly from previous runs, Selecta's mechanism for detecting changes in applications identifies the outlier.

5.4 Selecta Evaluation

Selecta's collaborative filtering approach is agnostic to the choice of applications and configurations. We evaluate Selecta for data analytics workloads on a subset of the cloud configuration space with the goal of understanding how to provision cloud storage for data analytics.

Instance	CPU cores	RAM (GB)	NVMe
`i3.xlarge`	4	30	1 x 950 GB
`r4.xlarge`	4	30	-
`i3.2xlarge`	8	60	1 x 1.9 TB
`r4.2xlarge`	8	60	-
`i3.4xlarge`	16	120	2 x 1.9 TB
`r4.4xlarge`	16	120	-

Table 5.2: AWS EC2 instance properties

Storage	Type	Locality	Use for Input/Output Data?	Use for Intermediate Data?
r-HDD	Block	Remote	✓	-
r-SSD	Block	Remote	✓	✓
l-NVMe	Block	Local	✓	✓
S3	Object	Remote	✓	-

Table 5.3: AWS storage options considered

5.4.1 Methodology

Cloud configurations: We deploy Selecta on Amazon EC2 and consider configurations with the instance and storage options shown in Tables 5.2 and 5.3. Among the possible VM and storage combinations, we consider seventeen candidate configurations. We trim the space to stay within our research budget and to focus on experiments that are most likely to uncover interesting insights about cloud storage for analytics. We choose EC2 instance families that are also supported by Databricks, a popular Spark-as-a-service provider [60]. `i3` is currently the only instance family available with NVMe Flash and `r4` instances allow for a fair comparison of storage options as they have the same memory to compute ratio. We only consider configurations where the intermediate data storage IOPS are equal to or greater than the input/output storage IOPS, as intermediate data has more random accesses. Since we find that most applications are I/O-bound with *r*-HDD, we only consider *r*-HDD for the instance size with the least amount of cores. We limit our analysis to *r*-HDD because our application datasets are up to 1 TB whereas instances with *l*-HDD on AWS come with a minimum of 6 TB disk storage, which would not be an efficient use of capacity. We do not consider local SAS/SATA SSDs as their storage capacity to CPU cores ratio is too low for most Spark workloads. We use Elastic Block Store (EBS) for remote block storage [18].

We use a cluster of 9 nodes for our evaluation. The cluster consists of one master node and eight executor nodes. The master node runs the Spark driver and YARN Resource Manager. Unless input/output data is stored in S3, we run a HDFS namenode on the master server as well. We configure framework parameters, such as the JVM heap size and number of executors, according to Spark tuning guidelines and match the number of executor tasks to the VM's CPU cores [55, 54].

Applications: We consider Spark [217] as a representative data analytics framework, similar

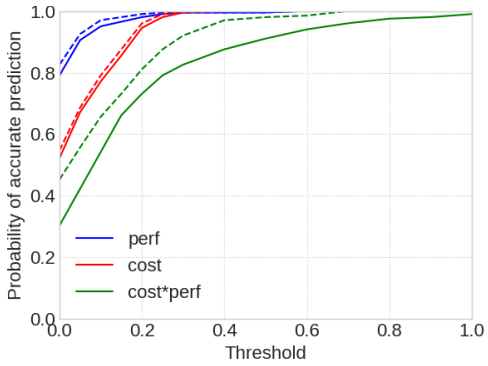
to previous studies [156, 200, 15]. We use Spark v2.1.0 and Hadoop v2.7.3 for HDFS. We evaluate Selecta with over one hundred Spark SQL and ML applications, each with two different dataset scales, for a total of 204 workloads. Our application set includes 92 queries of the TPC-DS benchmark with scale factors of 300 and 1000 GB [199]. We use the same scale factors for Spark SQL and ML queries from the TPC-BB (BigBench) benchmark which has structured, unstructured and semi-structured data modeled after the retail industry domain [82]. Since most BigBench queries are CPU-bound, we focus on eight queries which have more substantial I/O requirements: queries 3, 8, 14, 16, 21, 26, 28, 29. We also run 100 and 400 GB sort jobs [160]. Finally, we run a SQL equijoin query on two tables with 16M and 32M rows each and 4KB entries [161]. For all input and output files, we use the uncompressed Parquet data format [79].

Experiment methodology: We run each application on all candidate configurations to obtain the ground truth performance and optimal configuration choices for each application. To account for noise in the cloud we run each experiment (i.e., each application on each candidate configuration) three times and use the average across runs in our evaluation. Two runs are consecutive and one run is during a different time of day. We also validate our results by using data from one run as input to Selecta and the average performance across runs as the ground truth. To train and test Selecta, we use leave-one-out cross validation [173], meaning one workload at a time serves as the target application while the remaining workloads are used for training. We assume training applications are profiled on all candidate configurations, except for the sensitivity analysis in §5.4.4 where we investigate training matrix density requirements for accurate predictions.

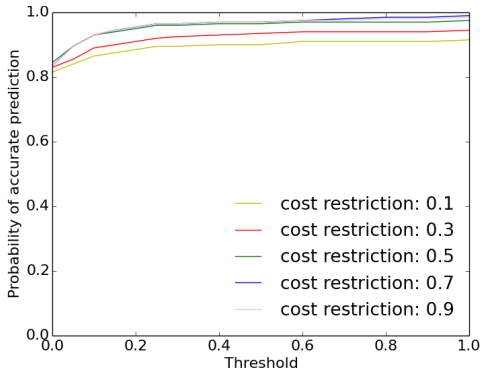
Metrics: We measure the quality of Selecta's predictions using two metrics. First, we report the relative root mean squared error (RMSE), a common metric for recommender systems. The second and more relevant metric for Selecta is the probability of making an accurate configuration recommendation. We consider a recommendation accurate if the configuration meets the user's cost-performance objective within a threshold T of the true optimal configuration for that application. For example, for a minimum cost objective with $T = 10\%$, the probability of an accurate prediction is the percentage of Selecta's recommendations (across all tested applications) whose true cost is within 10% of the true optimal cost configuration. Using a threshold is more robust to noise and allows us to make more meaningful conclusions about Selecta's accuracy, since a second-best configuration may have similar or significantly worse performance than the best configuration. Our performance metric is execution time and cost is in US dollars.

5.4.2 Prediction Accuracy

We provide a matrix with 204 rows as input to Selecta, where one row (application) is designated as the target application in each test round. We run Selecta 204 times, each time considering a different application as the target. For now, we assume all remaining rows of training data in the matrix are dense, implying the user has profiled training applications on all candidate configurations. The



(a) Probability of accurate recommendations within a threshold from optimal. Dotted lines are after one feedback iteration.



(b) Probability of accurate configuration recommendation for performance within threshold, given strict cost restrictions.

Figure 5.3: Probability of accurate recommendations within a threshold from optimal.

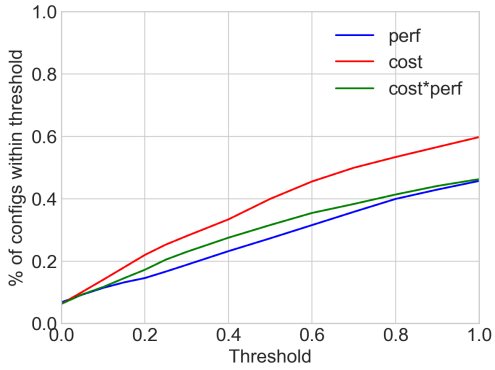


Figure 5.4: Percentage of configurations whose true performance, cost, and cost-performance is within the threshold. Across all of the applications, more than half of the configurations have performance that is over 100% higher than the true best configuration for each application.

single target application row is sparse, containing only two entries out of 17, one for each of the profiling runs on reference configurations.

Selecta predicts performance with a relative RMSE of 36%, on average across applications. To understand how Selecta's performance predictions translate into recommendations, we plot accuracy in Figure 5.3a for performance, cost and cost*performance objectives. The plot shows the probability of near-optimal recommendations as a function of the threshold T defining what percentage from optimal is considered close enough. When searching for the best performing configuration, Selecta has a 94% probability of recommending a configuration within 10% of optimal. For a minimum cost objective, Selecta has a 80% probability of recommending a configuration within 10% of optimal. Predicting cost*performance is more challenging since errors in Selecta's relative execution time predictions for an application across candidate configurations are squared: cost*performance = $(\text{execution\_time})^2$ * config_cost_per_hour.

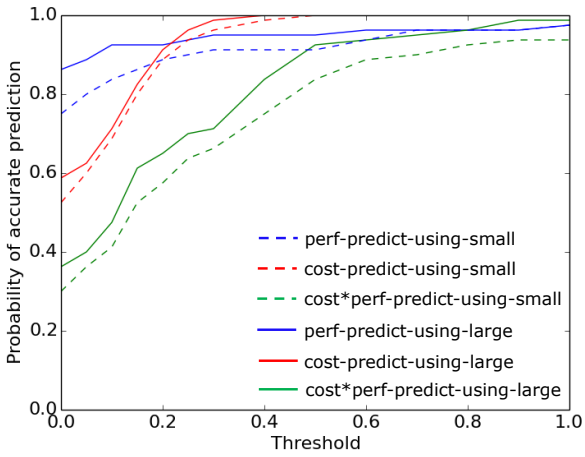


Figure 5.5: Selecta's accuracy with large datasets using predictions from small dataset vs. re-computing prediction with large datasets.

The dotted lines in Figure 5.3a show how accuracy improves after a single feedback round. Here, we assume the target application has the same dataset in the feedback round. This provides additional training input for the target application row (either a new entry if the recommended configuration was not a reference configuration, or a new sample to average to existing data if the recommended configuration was a reference configuration). The probability of near-optimal recommendations increases most noticeably for the cost*performance objective, from 52% to 65% after feedback, with T=10%.

Figure 5.3b shows the probability of accurate recommendations for objectives of the form "select the best performing configuration given a fixed cost restriction C." For this objective, we consider Selecta's recommendation accurate if its cost is less than or equal to the budget and if its performance is within the threshold of the true best configuration for the objective. Selecta achieves between 83% and 94% accuracy for the cost restrictions in Figure 5.3b assuming T=10%. The long tail is due to performance prediction errors that lead Selecta to underestimate the execution cost for a small percentage of configurations (i.e., cases where Selecta recommends a configuration that is actually over budget).

Figure 5.4 shows the distribution of the true performance, cost, and performance-cost data. The figure plots the percentage of configurations that fall within the threshold. Across all of the applications, more than half of the configurations have performance that is over 100% higher than the true best configuration for each application. This highlights how important and challenging it is to select the right configuration for a job.

In Figure 5.6, we compare Selecta's accuracy against four baselines. The first baseline is a random forest predictor, similar to the approach used by PARIS [214]. We use the following features: the number of CPU cores, disk IOPS and disk MB/s the configuration provides, the intermediate and

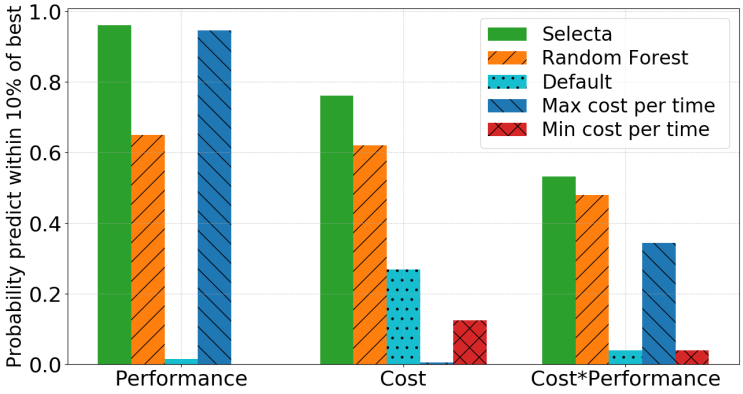


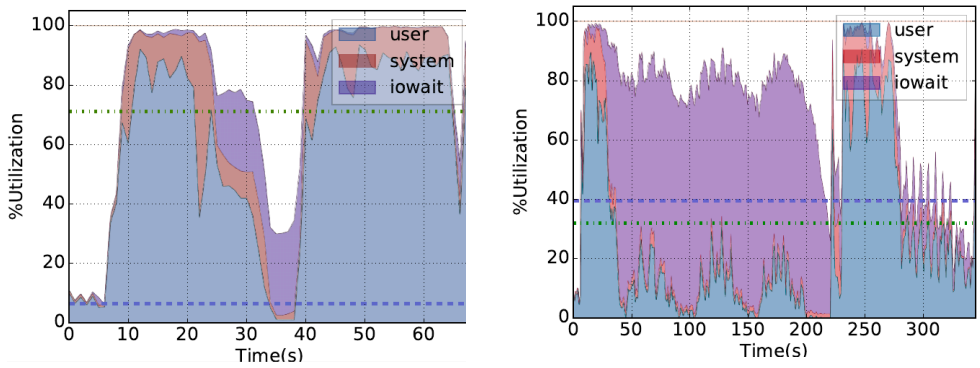
Figure 5.6: Selecta's accuracy compared to baseline approaches.

input/output data capacity of the application, and the CPU utilization, performance, and total disk throughput measured when running the application on each of the two reference configurations. Although the random forest predictor leverages more features than Selecta, it has lower accuracy. Collaborative filtering is a better fit for the sparse nature of the training data. We find the most important features in the random forest model are all related to I/O (e.g., the I/O throughput measured when running the application on the reference configurations and the read/write IOPS supported by the storage used for intermediate data), which emphasizes the importance of selecting the right storage.

The second baseline (labeled 'default') in Figure 5.6 uses the recommended default configurations documented in Databricks engineering blog posts: *l*-NVMe for intermediate data and S3 for input/output data [61, 63, 62]. The 'max cost per time' baseline uses the simple heuristic of always picking the most expensive instance per unit time. The 'min cost per time' baseline chooses the least expensive instance per unit time. Selecta outperforms all of these heuristic strategies, confirming the need for a tool to automate configuration selection.

5.4.3 Evolving Datasets

We study the impact of dataset size on application performance and Selecta's predictions using the small and large dataset scales described in §5.4.1. We train Selecta using all 102 workloads with small datasets, then evaluate Selecta's prediction accuracy for the same workloads with large datasets. The dotted lines in Figure 5.5 plots Selecta's accuracy when recommending configurations for applications with large datasets solely based on profiling runs of the application with a smaller dataset. The solid lines show accuracy when Selecta re-profiles applications with large datasets to make predictions. For approximately 8% of applications, profiling runs with small datasets are not sufficient indicators of performance with large datasets.



(a) Query on 300GB is CPU-bound.

(b) Query on 1TB is IO-bound.

Figure 5.7: CPU utilization over time for TPC-DS query 89 on `r4.xlarge` cluster with r-SSD. For this query, performance with a small dataset is not indicative of performance with a larger dataset. Selecta detects difference in average iowait percentage (blue dotted line).

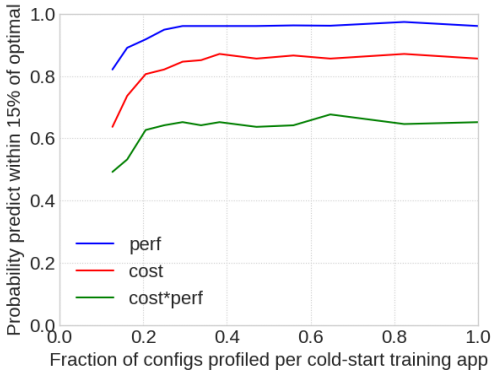
We find that in cases where the performance with a small dataset is not indicative of performance with a large dataset, the relationship between compute and I/O intensity of the application is affected by the dataset size. As described in §5.3.3, Selecta detects these situations by comparing CPU utilization statistics for the small and large dataset runs. Figure 5.7 shows an example of a workload for which small dataset performance is not indicative of performance with a larger dataset. We use the Intel Performance Analysis Tool to record and plot CPU utilization [106]. When the average iowait percentage for the duration of the run changes significantly between the large and small profiling runs on the reference configuration, it is generally best to profile the application on the reference configurations and treat it as a new application.

5.4.4 Sensitivity Analysis

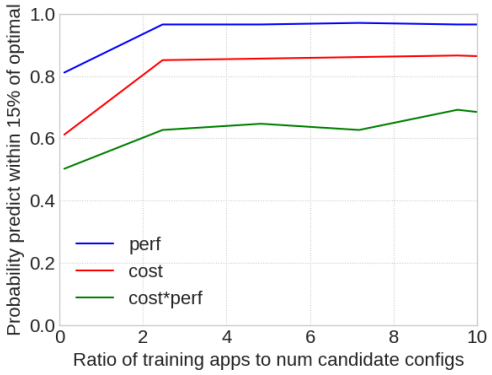
We perform a sensitivity analysis to determine input matrix density requirements for accurate predictions. We look at both the density of matrix rows (i.e., the percentage of candidate configurations that training applications are profiled on) and the density of matrix columns (i.e., the number of training applications used). We also discuss sensitivity to the choice of reference configurations.

Figure 5.8a shows how Selecta's accuracy for performance, cost and cost*performance objectives varies as a function of input matrix density. Assuming 203 training applications have accumulated in the system over time, we show that, on average across target applications, rows only need to be approximately 20 to 30% dense for Selecta to achieve sufficient accuracy. This means that at steady state, users should profile training applications on about 20-30% of the candidate configurations (including reference configurations). Profiling additional configurations has diminishing returns.

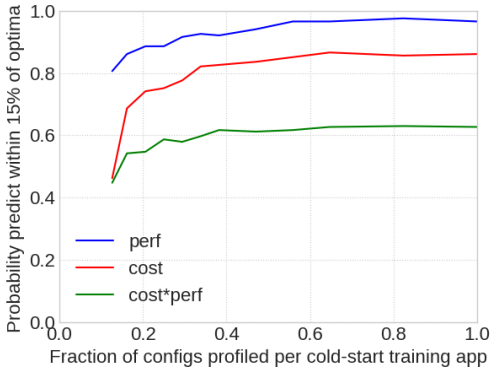
We consider a cold start situation in which a user wants to jump start the system by profiling a



(a) Sensitivity to input matrix density in *steady state*: 20% density per row suffices for accurate predictions.



(b) Sensitivity to number of training applications, profiled on all configurations: 2.5× the number of configs suffices.



(c) Sensitivity to input matrix density for *cold start*: ∼50% density per row (training application) required.

Figure 5.8: Sensitivity analysis: accuracy as a function of input matrix density

limited set of training applications across all candidate configurations. Figure 5.8b shows the number of training applications required to achieve desired accuracy. Here, for each target application testing round, we take the 203 training applications we have and randomly remove a fraction of the rows (training applications). We ensure to drop the row corresponding to the different dataset scale factor run of the target application, to ensure Selecta's accuracy does not depend on a training application directly related to the target application. Since the number of training applications required to achieve desirable accuracy depends on the size of the configuration space a user wishes to explore, the x-axis in Figure 5.8b represents the ratio of the number of training applications to the number of candidate configurations, R. We find that to jump start Selecta with dense training data from a cold start, users should provide 2.5× more training applications than the number of candidate configurations to achieve desirable accuracy. In our case, jump starting Selecta with more than $43 = \lceil 2.5 \times 17 \rceil$ training applications profiled on all 17 configurations has diminishing returns.

Naturally, a cold start requires more profiling data per row than the steady state scenario. Figure 5.8c plots accuracy as we vary the percentage of candidate configurations on which the training applications are profiled for a cold start using R=2.5 (i.e., 43 training applications for 17 candidate configurations). The figure shows that it is sufficient for users to profile the initial training applications on 40% to 60% of candidate configurations, as long as the configurations are selected randomly and independently for each job. As Selecta continues running and accumulates more training applications, the percentage of configurations users need to profile for training applications drops to 20-30% (this is the steady state result from Figure 5.8a).

We experimented with different reference configurations for Selecta. We find that accuracy is not very sensitive to the choice of references. We saw a slight benefit using references that have different VM and storage types. Although one reference configuration must remain fixed across all application runs since it is used to normalize performance, we found that the reference configuration used for the second profiling run could vary without significant impact on Selecta's accuracy.

5.5 Cloud Storage Insights

Our analysis of cloud configurations for data analytics reveals some insights for cloud storage. We discuss key takeaways and their implications for future research on storage systems.

NVMe storage is performance and cost efficient for data analytics: We find that configurations with NVMe Flash tend to offer not only the best performance, but also, more surprisingly, the lowest cost. Although NVMe Flash is the most expensive type of storage per GB/hr, its high bandwidth allows applications to run significantly faster, reducing the overall job execution cost.

On average across applications, we observe that l-NVMe Flash reduces job completion time of applications by 27% compared to r-SSD and 75% compared to r-HDD. Although we did not consider l-SSD or l-HDD configurations in our evaluation, we validate that local versus remote access to HDD and SSD achieves similar performance since our instances have sufficient network bandwidth (up to 10 Gb/s) and modern networking adds little overhead on top of HDD and SSD access latency [25]. In contrast, a previous study of Spark applications by Ousterhout et al. concluded that optimizing or eliminating disk accesses can only reduce job completion time by a median of at most 19% [156]. We believe the main reason for the increased impact of storage on end-to-end application performance is due to the newer version of Spark we use in our study (v2.1.0 versus v1.2.1). Spark has evolved with numerous optimizations targeting CPU efficiency, such as cache-aware computations, code generation for expression evaluation, and serialization [59]. With ongoing work in optimizing the CPU cycles spent on data analytics computations, for example by optimizing the I/O processing path [197], we expect the choice of storage to be of even greater importance.

The need for end-to-end optimization: In our experiments, remote HDD storage performed poorly, despite its cost effectiveness for long-living input/output data and its ability to match the

sequential bandwidth offered by SSD. Using the Linux `blktrace` tool [112] to analyze I/O requests at the block device layer, we found that although each Spark task reads/writes input/output data sequentially, streams from multiple tasks running on different cores interleave at the block device layer. Thus, the access stream seen by a remote HDD volume consists of approximately 60% random I/O operations, dramatically reducing performance compared to fully sequential I/O. This makes solutions with higher throughput for random accesses (e.g., using multiple HDDs devices or Flash storage) more appropriate for achieving high performance in data analytics. Increasing random I/O performance comes at a higher cost per unit time. In addition to building faster storage systems, we should attempt to optimize throughout the stack for sequential accesses when these accesses are available at the application level. Of course, there will always be workloads with intrinsically random access patterns that will not benefit from such optimizations.

5.6 Discussion

Our work focused on selecting storage configurations based on their performance and cost. Other important considerations include durability, availability, and consistency, particularly for long-term input/output data storage [126]. Developers may also prefer a particular storage API (e.g., POSIX files vs. object interface). Users can use these qualitative constraints to limit the storage space Selecta considers. Users may also choose different storage systems for high performance processing versus long term storage of important data.

Our study showed that separating input/output data and intermediate data uncovers a richer configuration space and allows for better customization of storage resources to the application requirements. We can further divide intermediate data into finer-grained streams such as shuffle data, broadcast data, and cached RDDs spilled from memory. Understanding the characteristics of these finer grain streams and how they should be mapped to storage options in the cloud may reveal further benefits.

Compression schemes offer an interesting trade-off between processing, networking, and storage requirements. In addition to compressing input/output files, systems like Spark allow compressing individual intermediate data streams using a variety of compression algorithms (lz4, lzf, and snappy) [190]. In future work, we plan to extend Selecta to consider compression options in addition to storage and instance configuration.

We used Selecta to optimize data analytics applications as they represent a common class of cloud workloads. Selecta's approach should be applicable to other data-intensive workloads too, as collaborative filtering does not make any specific assumptions about the application structure. In addition to considering other types of workloads, in future work, we will consider scenarios in which multiple workloads share cloud infrastructure. Delimitrou et al. have shown that collaborative filtering can classify application interference sensitivity (i.e., how much interference an application

will cause to co-scheduled applications and how much interference it can tolerate itself) [67, 68]. We also believe Selecta's collaborative filtering approach can be extended to help configure isolation mechanisms that limit interference between workloads, particularly on shared storage devices like NVMe which exhibit dramatically different behavior as the read-write access patterns vary [122].

We have presented Selecta as a tool that helps cloud users select the right virtual machine and storage configuration for their applications among the many options on traditional cloud computing platforms. Selecta's ability to automate resource configuration decisions at scale is also critical for cloud providers, particularly in the context of emerging cloud computing platforms. In managed cloud services, such as serverless computing, resource allocation becomes the responsibility of the cloud provider. Selecta can be used to learn across serverless job invocations and optimize resource allocation and task scheduling decisions for performance and cost objectives on serverless computing platforms. We also believe the collaborative filtering approach used in Selecta can be used to enhance the Pocket storage system discussed in Chapter 4. While Pocket currently relies on hints to learn job characteristics and make cost-effective resource allocations, Selecta can learn such characteristics across job invocations and automate the resource allocation process, alleviating users from the burden of providing hints.

5.7 Related Work

Selecting cloud configurations: Several recent systems unearth near-optimal cloud configurations for target workloads. CherryPick uses Bayesian Optimization to build a performance model that is just accurate enough to distinguish near-optimal configurations [15]. Model input comes solely from profiling the target application across carefully selected configurations. Ernest predicts performance for different VM and cluster sizes, targeting machine learning analytics applications [204]. PARIS takes a hybrid online/offline approach, using random forests to predict application performance on various VM configurations based on features such as CPU utilization obtained from profiling [214]. These systems do not consider the vast storage configuration options in the cloud nor the heterogeneous data streams of analytics applications which can dramatically impact performance.

Resource allocation with collaborative filtering: Our approach for predicting performance is most similar to Quasar [68] and Paragon [67], which apply collaborative filtering to schedule incoming applications on shared clusters. ProteusTM [71] applies collaborative filtering to auto-tune a transactional memory system. While these systems consider resource heterogeneity, they focus on CPU and memory. While Selecta applies a similar modeling approach, our exploration of the cloud storage configuration space is novel and reveals important insights.

Automating storage configurations: Many previous systems provide storage configuration recommendations [26, 195, 14, 149, 17, 95, 119]. Our work analyzes the trade-offs between traditional block storage and object storage available in the cloud. We also considering how heterogeneous

streams in data analytics applications should be mapped to heterogeneous storage options.

Analyzing performance of analytics frameworks: While previous studies analyze how CPU, memory, network and storage resources affect Spark performance [156, 200, 197, 134], our work is the first to evaluate the impact of new cloud storage options (e.g., NVMe Flash) and provide a tool to navigate the diverse storage configuration space.

Tuning application parameters: Previous work auto-tunes data analytics framework parameters such as the number of executors, JVM heap size, and compression schemes [101, 216, 215]. Our work is complementary. Users set application parameters and then run Selecta to obtain a near-optimal hardware configuration.

5.8 Conclusion

The large and increasing number of storage and compute options on cloud services makes configuring data analytics clusters for high performance and cost efficiency difficult. We presented Selecta, a tool that learns near-optimal configurations of compute and storage resources based on sparse training data collected across applications and candidate configurations. Requiring only two profiling runs of the target application, Selecta predicts near-optimal performance configurations with 94% probability and near-optimal cost configurations with 80% probability. Moreover, Selecta allowed us to analyze cloud storage options for data analytics and reveal important insights, including the cost benefits of NVMe Flash storage, the need for fine-gain allocation of storage capacity and bandwidth in the cloud, and the need for cross-layer storage optimizations. We believe that, as data-intensive workloads grow in complexity and cloud options for compute and storage increase, tools like Selecta will become increasingly useful for end users, systems researchers, and even cloud providers (e.g., for scheduling 'serverless' application code).

Chapter 6

Conclusion

This dissertation has shown how to build high performance, cost-effective, and easy-to-use cloud storage systems. We addressed two critical requirements. First, we enabled *fast, predictable access to remote data* on modern Flash storage. This ensures that storage allocation is not constrained by the physical characteristics of server hardware and resource allocations can be customized to the requirements of individual applications. Second, we automated storage resource management with *intelligent allocation of storage resources*. We demonstrated how to effectively learn application characteristics through hints or a machine learning model trained on past performance data to make resource-efficient storage allocation decisions at scale. We built three new systems that collectively improve the performance and resource efficiency of cloud storage, which we summarize below.

ReFlex is a custom network-storage operating system that enables applications to access remote Flash over commodity cloud networks to improve resource allocation flexibility without sacrificing the performance that applications achieve with local Flash. ReFlex processes requests to completion, adaptively batches requests, and avoids unnecessary data copies to achieve 11× higher throughput per core compared to a Linux system that uses `libaio-libevent` for remote access to Flash storage. ReFlex also introduces a novel I/O scheduler to enforce tail latency and throughput guarantees for up to thousands of tenants sharing a remote Flash storage device.

Pocket is a distributed ephemeral data store that builds on top of ReFlex to provide application-aware storage resource allocation along with low latency and high throughput access to data. We specifically designed Pocket to efficiently share intermediate data in a new class of workloads, called serverless analytics. Pocket features a new storage API that enables users or application frameworks to pass hints about application characteristics which Pocket's controller uses to make cost-effective resource allocations. The system places data across multiple storage technology tiers to maximize performance and cost efficiency. Finally, Pocket continuously monitors cluster resource utilization to autoscale resources based on application load. The system achieves comparable performance to a popular in-memory data store, Redis, while saving close to 60% in cost.

Selecta is a system that automates resource allocation decisions for recurring jobs in the cloud without requiring user or application framework hints. The system recommends a near-optimal resource allocation for a target application by predicting the jobs execution time across a variety of cloud resource configurations based on past performance data collected across runs. Using collaborative filtering, Selecta makes fast, accurate recommendations with sparse training data by uncovering latent similarities across jobs and resource configurations. Selecta can find resource allocations that minimize the execution time of a job, minimize the cost of running a job, or maximize the performance of a job within a budget. For minimizing job execution time, Selecta has a 94% probability of recommending a configuration that performs within 10% of the true highest performance configuration.

The contributions of this dissertation — fast access to remote data, elastic distributed storage, and automated resource allocation decisions — are vital in the context of the shift that cloud computing is undergoing today. Stimulated by an exponential growth in the volume of data and non-expert users who desire higher levels of abstraction for resource management, cloud providers are increasingly offering fully-managed compute and storage services. Serverless computing (e.g., AWS Lambda [23]) and machine learning as a service platforms (e.g., Google Cloud AI [89]) are examples of such services. As cloud computing evolves from users managing virtual machines with fixed resource ratios to cloud providers increasingly managing resources automatically for user tasks, insights from the design and implementation of ReFlex, Pocket, and Selecta are valuable in enabling high performance, resource-efficient, and easy-to-use cloud storage.

6.1 Future Directions

Our work also opens several interesting avenues for future research. As cloud applications evolve and datacenter hardware becomes more heterogeneous, there are many opportunities to innovate across layers — from application frameworks to hardware interfaces — to bridge the requirements of applications with the capabilities of modern hardware.

Storage for machine learning: Our current work has focused on addressing the cloud storage requirements of multi-tier web applications, data analytics workloads, and serverless analytics jobs. Machine learning workloads are a new class of applications increasingly being deployed on cloud platforms and imposing new, challenging requirements for data management [138]. Processing massive datasets with low latency and high throughput requires striking the right balance between data disaggregation (to support large scale) and data locality (to maximize performance). ReFlex provides fast access to remote data, enabling machine learning models to train on arbitrarily large, distributed datasets with high parallelism. However, optimizing locality is still desirable as massive datasets consume significant power and network bandwidth to move. Fortunately, the access patterns of machine learning workloads and trends in emerging storage devices present opportunities

to optimize data movement by pushing some computation closer to the data. For example, many machine learning workloads involve updating only a fraction of weights in a weight vector stored on a remote node. By leveraging on-chip processing capabilities in new Flash storage devices (e.g., SmartSSDs [174]), we can perform filter operations at storage nodes and send only the data of interest to remote nodes for further processing. There are many open questions to explore, including how to program 'smart storage' devices, which computations to support for machine learning workloads, and how to coordinate distributed data processing in clusters of smart storage devices.

Opportunistic cloud storage: We saw that disaggregating compute and storage resources improves resource utilization by allowing resource allocations to be customized to individual application requirements. However, cloud providers are still motivated to overprovision resources to have slack for accommodating unanticipated load spikes. Hence, a promising way to maximize resource utilization is to opportunistically run storage services on slack resources [47]. By harvesting temporarily idle resources, cloud providers can offer storage services to users at significantly lower cost. A major challenge is enabling higher priority services to easily reclaim slack resources when needed, while still enabling the opportunistic storage services to store and process data with reasonable performance guarantees on unreliable infrastructure [220, 148]. Pocket is a good candidate for an opportunistic storage service since the system targets ephemeral data, which has low durability requirements. However, Pocket's original design does not provide fault tolerance as application frameworks are responsible for relaunching tasks to regenerate any data that gets lost. Abstracting potentially frequent resource revocations from users is important for enabling Pocket to run as an opportunistic storage service on slack resources. The opportunistic storage service must make important decisions about when and where to replicate and/or checkpoint data. These decisions can be greatly informed by training a machine learning model to predict slack resource usage trends over time and provide early warning signals of resource revocations.

Adaptive fault tolerance: Our work on distributed storage for serverless analytics focused on the broad class of temporary data generated and exchanged between intermediate stages of a job. However, not all temporary data within a job has the same value. The data generated and consumed during job execution varies greatly in terms of its fault tolerance requirements. For example, in a long-running job, the temporary data generated by tasks near the final stage is likely more valuable than the data generated by tasks early on in the job. This is because regenerating the final stage data may require re-launching tasks from all stages prior to when a fault occurred. In this example, we would likely want our storage system to checkpoint temporary data with increasing frequency as a job progresses. More generally, co-designing application frameworks and storage systems to automatically infer the value of individual data objects in a job and translate this value into a fault tolerance scheme for each object is an interesting direction for future work. One approach is to leverage a job's execution graph, which captures data dependency chains, to extract the value of individual data objects.

Bibliography

[1] Apache Storm. `https://storm.apache.org`, 2015.

[2] IX-project: protected dataplane for low latency and high performance. `https://github.com/ix-project`, 2016.

[3] Apache CouchDB. `http://couchdb.apache.org`, 2018.

[4] Apache Crail (incubating). `http://crail.incubator.apache.org`, 2018.

[5] Apache openwhisk (incubating). `https://openwhisk.apache.org`, 2018.

[6] gg: The Stanford Builder. `https://github.com/stanfordsnr/gg`, 2018.

[7] Horizontal Pod Autoscaler. `https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale`, 2018.

[8] Kubernetes operations (kops). `https://github.com/kubernetes/kops`, 2018.

[9] Kubernetes: Production-Grade Container Orchestration. `https://kubernetes.io`, 2018.

[10] Memcached – a distributed memory object caching system. `https://memcached.org`, 2018.

[11] Michael Abd-El-Malek, William V. Courtright II, Chuck Cranor, Gregory R. Ganger, James Hendricks, Andrew J. Klosterman, Michael P. Mesnier, Manish Prasad, Brandon Salmon, Raja R. Sambasivan, Shafeeq Sinnamohideen, John D. Strunk, Eno Thereska, Matthew Wachs, and Jay J. Wylie. Ursa minor: Versatile cluster-based storage. In *Proc. of the FAST Conference on File and Storage Technologies*, 2005.

[12] Sameer Agarwal, Srikanth Kandula, Nicolas Bruno, Ming-Chuan Wu, Ion Stoica, and Jingren Zhou. Re-optimizing data-parallel computing. In *Proc. of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 21–21, 2012.

[13] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D. Davis, Mark S. Manasse, and Rina Panigrahy. Design tradeoffs for ssd performance. In *Proc. of the USENIX Annual Technical Conference*, ATC'08, pages 57–70, 2008.

[14] Christoph Albrecht, Arif Merchant, Murray Stokely, Muhammad Waliji, Francois Labelle, Nathan Coehlo, Xudong Shi, and Eric Schrock. Janus: Optimal flash provisioning for cloud storage workloads. In *Proc. of the USENIX Annual Technical Conference*, ATC'13, pages 91–102, 2013.

[15] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. CherryPick: Adaptively unearthing the best cloud configurations for big data analytics. In *Proc. of the 14th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'17, pages 469–482, 2017.

[16] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics. In *Proc. of the USENIX Symposium on Networked Systems Design and Implementation*, NSDI'17, pages 469–482, 2017.

[17] Guillermo A. Alvarez, Elizabeth Borowsky, Susie Go, Theodore H. Romer, Ralph Becker-Szendy, Richard Golding, Arif Merchant, Mirjana Spasojevic, Alistair Veitch, and John Wilkes. Minerva: An automated resource provisioning tool for large-scale storage systems. *ACM Transactions on Computer Systems*, 19(4):483–518, November 2001.

[18] Amazon. Amazon elastic block store (EBS). `https://aws.amazon.com/ebs`, 2018.

[19] Amazon. Amazon ElastiCache. `https://aws.amazon.com/elasticache`, 2018.

[20] Amazon. Amazon redshift. `https://aws.amazon.com/redshift`, 2018.

[21] Amazon. Amazon S3 reduced redundancy storage. `https://aws.amazon.com/s3/reduced-redundancy`, 2018.

[22] Amazon. Amazon simple storage service. `https://aws.amazon.com/s3`, 2018.

[23] Amazon. AWS lambda. `https://aws.amazon.com/lambda`, 2018.

[24] Amazon. AWS lambda limits. `https://docs.aws.amazon.com/lambda/latest/dg/limits.html`, 2018.

[25] Ganesh Ananthanarayanan, Ali Ghodsi, Scott Shenker, and Ion Stoica. Disk-locality in datacenter computing considered irrelevant. In *Proc. of USENIX Hot Topics in Operating Systems*, HotOS'13, pages 12–12, 2011.

[26] Eric Anderson, Michael Hobbs, Kimberly Keeton, Susan Spence, Mustafa Uysal, and Alistair Veitch. Hippodrome: Running circles around storage administration. In *Proc. of the 1st USENIX Conference on File and Storage Technologies*, FAST'02, pages 13–13, 2002.

[27] Sebastian Angel, Hitesh Ballani, Thomas Karagiannis, Greg O'Shea, and Eno Thereska. End-to-end performance isolation through virtual datacenters. In *Proc. of USENIX Operating Systems Design and Implementation*, OSDI'14, pages 233–248, October 2014.

[28] Masoud Saeida Ardekani and Douglas B. Terry. A self-configurable geo-replicated cloud storage system. In *Proc. of the 11th USENIX Symposium on Operating Systems Design and Implementation*, OSDI'14, pages 367–381, 2014.

[29] Dan Ardelean, Amer Diwan, and Chandra Erdman. Performance analysis of cloud applications. In *Proc. of the 15th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'18, pages 405–417, 2018.

[30] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Communications ACM*, 53(4):50–58, April 2010.

[31] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proc. of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, pages 53–64, 2012.

[32] Avago Technologies. Storage and PCI Express – A Natural Combination. `http://www.avagotech.com/applications/datacenters/enterprise-storage`, 2016.

[33] Jens Axboe. Linux block IOpresent and future. In *Ottawa Linux Symp*, pages 51–61, 2004.

[34] Microsoft Azure. Storage. `https://azure.microsoft.com/en-us/services/storage/`, 2016.

[35] Luiz Andr Barroso, Jimmy Clidaras, and Urs Hlzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, Second Edition.* 2013.

[36] Adam Belay. *Unleashing hardware potential through better operating system abstractions.* PhD thesis, Stanford University, 2016.

[37] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. Dune: Safe user-level access to privileged cpu features. In *Proc. of USENIX Operating Systems Design and Implementation*, OSDI'12, pages 335–348, 2012.

[38] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. IX: A protected dataplane operating system for high throughput and low latency. In *Proc. of USENIX Operating Systems Design and Implementation*, OSDI'14, pages 49–65, October 2014.

[39] Robert Bell, Yehuda Koren, and Chris Volinsky. Modeling relationships at multiple scales to improve accuracy of large recommender systems. In *Proc. of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '07, pages 95–104, 2007.

[40] Robert M. Bell, Yehuda Koren, and Chris Volinsky. The BellKor 2008 Solution to the Netflix Prize. Technical report, 2008.

[41] R. Bez, E. Camerlenghi, A. Modelli, and A. Visconti. Introduction to flash memory. *Proc. of the IEEE*, 91(4):489–502, April 2003.

[42] Matias Bjørling, Jens Axboe, David Nellans, and Philippe Bonnet. Linux block io: introducing multi-queue ssd access on multi-core systems. In *Proc. of International Systems and Storage Conference*, page 22. ACM, 2013.

[43] Matias Bjørling, Javier Gonzalez, and Philippe Bonnet. Lightnvm: The linux open-channel SSD subsystem. In *Proc. of 15th USENIX Conference on File and Storage Technologies*, FAST'17, pages 359–374, 2017.

[44] Simona Boboila and Peter Desnoyers. Write endurance in flash drives: Measurements and analysis. In *Proc. of USENIX Conference on File and Storage Technologies*, FAST'10, pages 9–9, 2010.

[45] Léon Bottou. *Large-Scale Machine Learning with Stochastic Gradient Descent*, pages 177–186. Physica-Verlag HD, 2010.

[46] John Bruno, Jose Brustoloni, Eran Gabber, Banu Ozden, and Abraham Silberschatz. Disk scheduling with quality of service guarantees. In *Proc. of the IEEE International Conference on Multimedia Computing and Systems - Volume 2*, ICMCS '99, pages 400–405. IEEE Computer Society, 1999.

[47] Marcus Carvalho, Walfredo Cirne, Francisco Brasileiro, and John Wilkes. Long-term SLOs for reclaimed cloud computing resources. In *Proc. of the ACM Symposium on Cloud Computing*, SOCC '14, pages 20:1–20:13, 2014.

[48] Adrian M. Caulfield and Steven Swanson. QuickSAN: A storage area network for fast, distributed, solid state disks. In *Proc. of International Symposium on Computer Architecture*, ISCA '13, pages 464–474. ACM, 2013.

[49] Mallikarjun Chadalapaka, Hemal Shah, Uri Elzur, Patricia Thaler, and Michael Ko. A study of iSCSI extensions for RDMA (iSER). In *Proc. of ACM SIGCOMM Workshop on Network-I/O Convergence: Experience, Lessons, Implications*, NICELI '03, pages 209–219. ACM, 2003.

[50] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. In *Proc. of USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, OSDI'06, pages 205–218, 2006.

[51] Jeffrey S. Chase, Darrell C. Anderson, Prachi N. Thakar, Amin M. Vahdat, and Ronald P. Doyle. Managing energy and server resources in hosting centers. In *Proc. of the Eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01, pages 103–116, 2001.

[52] Chelsio Communications. NVM Express over Fabrics. `http://www.chelsio.com/wp-content/uploads/resources/NVM_Express_Over_Fabrics.pdf`, 2014.

[53] Gong Chen, Wenbo He, Jie Liu, Suman Nath, Leonidas Rigas, Lin Xiao, and Feng Zhao. Energy-aware server provisioning and load dispatching for connection-intensive internet services. In *Proc. of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'08, pages 337–350, 2008.

[54] Cloudera. How-to: Tune your apache spark jobs. `https://blog.cloudera.com/blog/2015/03/how-to-tune-your-apache-spark-jobs-part-2/`, 2015.

[55] Cloudera. Tuning spark applications. `https://www.cloudera.com/documentation/enterprise/5-9-x/topics/admin_spark_tuning.html`, 2017.

[56] Franois Alexandre Colombani. HDD, SSHD, SSD or PCIe SSD. Storage Newsletter, `http://www.storagenewsletter.com/rubriques/market-reportsresearch/hdd-sshd-ssd-or-pcie-ssd/`, 2015.

[57] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *Proc. of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 153–167, 2017.

[58] Benoit Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Q. Munir, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, and Philipp Unterbrunner. The snowflake elastic data warehouse. In *Proc. of the International Conference on Management of Data*, SIGMOD '16, pages 215–226, 2016.

[59] Databricks. Project tungsten: Bringing apache spark closer to bare metal. `https://databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal.html`, 2015.

[60] Databricks. AWS configurations for Spark. `https://docs.databricks.com/user-guide/clusters/aws-config.html#ebs-volumes`, 2016.

[61] Databricks. Supported instance types. `https://databricks.com/product/pricing/instance-types`, 2016.

[62] Databricks. Accelerating workflows on databricks. `https://databricks.com/blog/2017/10/06/accelerating-r-workflows-on-databricks.html`, 2017.

[63] Databricks. Benchmarking big data sql platforms in the cloud. `https://databricks.com/blog/2017/07/12/benchmarking-big-data-sql-platforms-in-the-cloud.html`, 2017.

[64] Databricks. Databricks serverless: Next generation resource management for Apache Spark. `https://databricks.com/blog/2017/06/07/databricks-serverless-next-generation-resource-management-for-apache-spark.html`, 2017.

[65] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications ACM*, 56(2):74–80, February 2013.

[66] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *Proc. of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 205–220, 2007.

[67] Christina Delimitrou and Christos Kozyrakis. Paragon: Qos-aware scheduling for heterogeneous datacenters. In *Proc. of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 77–88, 2013.

[68] Christina Delimitrou and Christos Kozyrakis. Quasar: Resource-efficient and QoS-aware cluster management. In *Proc. of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 127–144, 2014.

[69] Christina Delimitrou, Daniel Sánchez, and Christos Kozyrakis. Tarcil: reconciling scheduling speed and quality in large shared clusters. In *Proc. of the ACM Symposium on Cloud Computing*, SoCC'15, pages 97–110. ACM, 2015.

[70] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. In *Symposium Proceedings on Communications Architectures &Amp; Protocols*, SIGCOMM '89, pages 1–12. ACM, 1989.

[71] Diego Didona, Nuno Diegues, Anne-Marie Kermarrec, Rachid Guerraoui, Ricardo Neves, and Paolo Romano. Proteustm: Abstraction meets performance in transactional memory. In *Proc. of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 757–771, 2016.

[72] Ronald P. Doyle, Jeffrey S. Chase, Omer M. Asad, Wei Jin, and Amin M. Vahdat. Model-based resource provisioning in a web service utility. In *Proc. of the 4th USENIX Symposium on Internet Technologies and Systems*, USITS'03, pages 5–5, 2003.

[73] Adam Dunkels. Design and implementation of the lwip, 2001.

[74] Facebook Inc. Open Compute Project. `http://www.opencompute.org/projects`, 2015.

[75] Facebook Inc. RocksDB: A persistent key-value store for fast storage environments. `http://rocksdb.org`, 2015.

[76] Andrew D. Ferguson, Peter Bodik, Srikanth Kandula, Eric Boutin, and Rodrigo Fonseca. Jockey: Guaranteed job latency in data parallel clusters. In *Proc. of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 99–112, 2012.

[77] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. From laptop to lambda: Outsourcing everyday jobs to thousands of transient functional containers. In *USENIX Annual Technical Conference*, ATC'19, 2019.

[78] Sadjad Fouladi, Riad S. Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. Encoding, fast and slow: Low-latency video processing using thousands of tiny threads. In *Proc. of the 14th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'17, pages 363–376, 2017.

[79] Apache Software Foundation. Apache parquet. `https://parquet.apache.org/`, 2014.

[80] A. Gandhi, Yuan Chen, D. Gmach, M. Arlitt, and M. Marwah. Minimizing data center sla violations and power consumption via hybrid resource provisioning. In *Proc. of the 2011 International Green Computing Conference and Workshops*, IGCC '11, pages 1–8, 2011.

[81] Anshul Gandhi, Mor Harchol-Balter, Ram Raghunathan, and Michael A. Kozuch. Autoscale: Dynamic, robust capacity management for multi-tier data centers. *ACM Transactions on Computer Systems*, 30(4):14:1–14:26, November 2012.

[82] Ahmad Ghazal, Tilmann Rabl, Minqing Hu, Francois Raab, Meikel Poess, Alain Crolotte, and Hans-Arno Jacobsen. Bigbench: Towards an industry standard benchmark for big data analytics. In *Proc. of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 1197–1208, 2013.

[83] Sanjay Ghemawat and Jeff Dean. LevelDB. `https://github.com/google/leveldb`, 2014.

[84] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *Proc. of ACM Symposium on Operating Systems Principles*, SOSP '03, pages 29–43. ACM, 2003.

[85] Google. Google compute engine persistent disk. `https://cloud.google.com/persistent-disk`, 2017.

[86] Google. Introducing Cloud Dataflow Shuffle: For up to 5x performance improvement in data analytic pipelines. `https://cloud.google.com/blog/products/gcp/introducing-cloud-dataflow-shuffle-for-up-to-5x-performance-improvement-in-data-analytic-pipelines`, 2017.

[87] Google. Cloud functions. `https://cloud.google.com/functions`, 2018.

[88] Google. Google bigquery. `https://cloud.google.com/bigquery`, 2018.

[89] Google. Google Cloud AI. `https://cloud.google.com/products/ai/`, 2019.

[90] Jorge Guerra, Himabindu Pucha, Joseph Glider, Wendy Belluomini, and Raju Rangaswami. Cost effective storage using extent based dynamic tiering. In *Proc. of the 9th USENIX Conference on File and Stroage Technologies*, FAST'11, pages 20–20, 2011.

[91] Ajay Gulati, Irfan Ahmad, and Carl A. Waldspurger. PARDA: proportional allocation of resources for distributed storage access. In *Proc. of USENIX File and Storage Technologies*, FAST '09, pages 85–98, 2009.

[92] Ajay Gulati, Arif Merchant, Mustafa Uysal, Pradeep Padala, and Peter Varman. Efficient and adaptive proportional share I/O scheduling. *SIGMETRICS Perform. Eval. Rev.*, 37(2):79–80, October 2009.

[93] Ajay Gulati, Arif Merchant, and Peter J. Varman. pclock: An arrival curve based approach for qos guarantees in shared storage systems. In *Proc. of ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '07, pages 13–24. ACM, 2007.

[94] Ajay Gulati, Arif Merchant, and Peter J. Varman. mClock: handling throughput variability for hypervisor io scheduling. In *Proc. of USENIX Operating Systems Design and Implementation*, OSDI'10, pages 437–450, 2010.

[95] Ajay Gulati, Ganesha Shanmuganathan, Irfan Ahmad, Carl Waldspurger, and Mustafa Uysal. Pesto: Online storage performance management in virtualized datacenters. In *Proc. of the 2Nd ACM Symposium on Cloud Computing*, SOCC '11, pages 19:1–19:14, 2011.

[96] Pradeep Kumar Gunda, Lenin Ravindranath, Chandramohan A. Thekkath, Yuan Yu, and Li Zhuang. Nectar: Automatic management of data and computation in datacenters. In *Proc. of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 75–88, 2010.

[97] James Hamilton. Keynote: Internet-scale service infrastructure efficiency. In *Proc. of International Symposium on Computer Architecture*, ISCA'09, June 2009.

[98] Sangjin Han, Norbert Egi, Aurojit Panda, Sylvia Ratnasamy, Guangyu Shi, and Scott Shenker. Network support for resource disaggregation in next-generation datacenters. In *Proc. of ACM Workshop on Hot Topics in Networks*, HotNets-XII, pages 10:1–10:7. ACM, 2013.

[99] Nikolas Roman Herbst, Samuel Kounev, and Ralf Reussner. Elasticity in cloud computing: What it is, and what it is not. In *Proc. of the 10th International Conference on Autonomic Computing*, ICAC'13, pages 23–27, 2013.

[100] Herodotos Herodotou, Fei Dong, and Shivnath Babu. No one (cluster) size fits all: Automatic cluster sizing for data-intensive analytics. In *Proc. of the 2Nd ACM Symposium on Cloud Computing*, SOCC '11, pages 18:1–18:14, 2011.

[101] Herodotos Herodotou, Harold Lim, Gang Luo, Nedyalko Borisov, Liang Dong, Fatma Bilgen Cetin, and Shivnath Babu. Starfish: A self-tuning system for big data analytics. In *Proc. of the Conference on Innovative Data Systems Research*, CIDR'11, pages 261–272, 2011.

[102] HGST. LinkedIn scales to 200 million users with PCIe Flash storage from HGST. `https://www.hgst.com/sites/default/files/resources/LinkedIn-Scales-to-200M-Users-CS.pdf`, 2014.

[103] Tibor Horvath and Kevin Skadron. Multi-mode energy management for multi-tier server clusters. In *Proc. of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, pages 270–279, 2008.

[104] Cheng Huang, Huseyin Simitci, Yikang Xu, Aaron Ogus, Brad Calder, Parikshit Gopalan, Jin Li, and Sergey Yekhanin. Erasure coding in windows azure storage. In *Proc. of the USENIX Conference on Annual Technical Conference*, ATC'12, pages 2–2, 2012.

[105] Nicolas Hug. Surprise, a Python library for recommender systems. `http://surpriselib.com`, 2017.

[106] Intel. Performance analysis tool (PAT). `https://github.com/intel-hadoop/PAT`, 2016.

[107] Intel Corp. Intel Rack Scale Architecture Platform. `http://www.intel.com/content/dam/www/public/us/en/documents/guides/rack-scale-hardware-guide.pdf`, 2015.

[108] Intel Corp. Dataplane Performance Development Kit. `https://dpdk.org`, 2016.

[109] Intel Corp. Storage Performance Development Kit. `https://01.org/spdk`, 2016.

[110] Virajith Jalaparti, Hitesh Ballani, Paolo Costa, Thomas Karagiannis, and Ant Rowstron. Bridging the tenant-provider gap in cloud services. In *Proc. of the Third ACM Symposium on Cloud Computing*, SoCC '12, pages 10:1–10:14, 2012.

[111] Jens Axboe. FIO: Flexible I/O Tester. `https://github.com/axboe/fio`, 2015.

[112] Alan D. Brunelle Jens Axboe and Nathan Scott. blktrace man page. `https://linux.die.net/man/8/blktrace`, 2006.

[113] Eun Young Jeong, Shinae Woo, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. mTCP: A highly scalable user-level TCP stack for multicore systems. In *Proc. of USENIX Networked Systems Design and Implementation*, NSDI'14, pages 489–502, 2014.

[114] Wei Jin, Jeffrey S. Chase, and Jasleen Kaur. Interposed proportional sharing for a storage service utility. In *Proc. of the Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '04/Performance '04, pages 37–48. ACM, 2004.

[115] Abhijeet Joglekar, Michael E. Kounavis, and Frank L. Berry. A scalable and high performance software iSCSI implementation. In *Proc. of USENIX Conference on File and Storage Technologies - Volume 4*, FAST'05, pages 20–20, 2005.

[116] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. Occupy the cloud: distributed computing for the 99%. In *Proc. of the 2017 Symposium on Cloud Computing*, SOCC'17, pages 445–451, 2017.

[117] Svilen Kanev, Juan Pablo Darago, Kim M. Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David M. Brooks. Profiling a warehouse-scale computer. In *Proc. of Annual International Symposium on Computer Architecture, ISCA '15*, pages 158–169, 2015.

[118] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proc. of the Twenty-ninth Annual ACM Symposium on Theory of Computing*, STOC '97, pages 654–663, 1997.

[119] Kimberley Keeton, Cipriano Santos, Dirk Beyer, Jeffrey Chase, and John Wilkes. Designing for disasters. In *Proc. of the 3rd USENIX Conference on File and Storage Technologies*, FAST '04, pages 59–62, 2004.

[120] Osama Khan, Randal Burns, James Plank, William Pierce, and Cheng Huang. Rethinking erasure codes for cloud file systems: Minimizing i/o for recovery and degraded reads. In *Proc. of the 10th USENIX Conference on File and Storage Technologies*, FAST'12, pages 20–20, 2012.

[121] Ana Klimovic, Christos Kozyrakis, Eno Thereska, Binu John, and Sanjeev Kumar. Flash storage disaggregation. In *Proc. of European Conference on Computer Systems*, EuroSys '16, pages 29:1–29:15, 2016.

[122] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. Reflex: Remote flash == local flash. In *Proc. of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 345–359, 2017.

[123] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. Selecta: Heterogeneous cloud storage configuration for data analytics. In *Proc. of the USENIX Annual Technical Conference*, ATC'18, pages 759–773, 2018.

[124] Ana Klimovic, Yawen Wang, Christos Kozyrakis, Patrick Stuedi, Jonas Pfefferle, and Animesh Trivedi. Understanding ephemeral storage for serverless analytics. In *Proc. of the USENIX Annual Technical Conference (ATC'18)*, pages 789–794, 2018.

[125] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. Pocket: Elastic ephemeral storage for serverless analytics. In *Proc. of the Symposium on Operating Systems Design and Implementation*, OSDI'18, pages 427–444, 2018.

[126] Gali Kovacs. EBS, EFS, or Amazon S3: which is the best cloud storage system for you? `https://cloud.netapp.com/blog/ebs-efs-amazons3-best-cloud-storage-system`, 2017.

[127] Andrew Krioukov, Prashanth Mohan, Sara Alspaugh, Laura Keys, David Culler, and Randy H. Katz. Napsac: Design and implementation of a power-proportional web cluster. In *Proc. of the First ACM SIGCOMM Workshop on Green Networking*, Green Networking '10, pages 15–22, 2010.

[128] Redis Labs. Redis. `https://redis.io`, 2018.

[129] Mark Lantz. Tape storage mounts a comeback. *IEEE Spectrum*, 55(9):32–37, Sep. 2018.

[130] Edward K. Lee and Chandramohan A. Thekkath. Petal: Distributed virtual disks. In *Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VII, pages 84–92, 1996.

[131] Jure Leskovec and Andrej Krevl. SNAP datasets: Stanford large network dataset collection. 2015.

[132] Jure Leskovec, Anand Rajaraman, and Jeffrey David Ullman. *Mining of massive datasets*. Cambridge University Press, 2014.

[133] Jacob Leverich. Mutilate: High-Performance Memcached Load Generator. `https://github.com/leverich/mutilate`, 2014.

[134] Haoyuan Li, Ali Ghodsi, Matei Zaharia, Scott Shenker, and Ion Stoica. Tachyon: Reliable, memory speed storage for cluster computing frameworks. In *Proc. of the ACM Symposium on Cloud Computing*, SOCC '14, pages 6:1–6:15, 2014.

[135] Harold C. Lim, Shivnath Babu, and Jeffrey S. Chase. Automated control for elastic storage. In *Proc. of the 7th International Conference on Autonomic Computing*, ICAC '10, pages 1–10, 2010.

[136] LinkedIn Inc. Project Voldemort: A distributed key-value storage system. `http://www.project-voldemort.com/voldemort`, 2015.

[137] Charles Loboz. Cloud resource usage-heavy tailed distributions invalidating traditional capacity planning models. *Journal of Grid Computing*, 10(1):85–108, 2012.

[138] Liang Luo, Jacob Nelson, Luis Ceze, Amar Phanishayee, and Arvind Krishnamurthy. Parameter hub: A rack-scale parameter server for distributed deep neural network training. In *Proc. of the ACM Symposium on Cloud Computing*, SoCC '18, pages 41–54, 2018.

[139] Ilias Marinos, Robert N.M. Watson, and Mark Handley. Network stack specialization for performance. In *Proc. of ACM SIGCOMM*, SIGCOMM'14, pages 175–186, 2014.

[140] Omid Mashayekhi, Hang Qu, Chinmayee Shah, and Philip Levis. Execution templates: Caching control plane decisions for strong scaling of data analytics. In *Proc. of the 2017 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '17, pages 513–526, 2017.

[141] Menage, Paul. cgroups. `https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt`, 2004.

[142] Arif Merchant, Mustafa Uysal, Pradeep Padala, Xiaoyun Zhu, Sharad Singhal, and Kang G. Shin. Maestro: quality-of-service in large disk arrays. In *Proc. of International Conference on Autonomic Computing*, ICAC'11, pages 245–254, 2011.

[143] J. Metz, Amber Huffman, Steve Sardella, and Dave Mintrun. The performance impact of NVM Express and NVM Express over Fabrics. `http://www.nvmexpress.org/wp-content/uploads/NVMe-Webcast-Slides-20141111-Final.pdf`, 2015.

[144] Microsoft. Azure files. `https://azure.microsoft.com/en-us/services/storage/files`, 2017.

[145] Microsoft. Azure functions. `https://azure.microsoft.com/en-us/services/functions`, 2018.

[146] Microsoft Azure. Azure redis cache. `https://azure.microsoft.com/en-us/services/cache`, 2018.

[147] Microsoft Azure. SQL data warehouse. `https://azure.microsoft.com/en-us/services/sql-data-warehouse`, 2018.

[148] Pulkit A. Misra, Inigo Goiri, Jason Kace, and Ricardo Bianchini. Scaling distributed file systems in resource-harvesting datacenters. In *USENIX Annual Technical Conference*, ATC'17, pages 799–811, 2017.

[149] Dushyanth Narayanan, Eno Thereska, Austin Donnelly, Sameh Elnikety, and Antony Rowstron. Migrating server storage to ssds: Analysis of tradeoffs. In *Proc. of the 4th ACM European Conference on Computer Systems*, EuroSys '09, pages 145–158, 2009.

[150] Netflix. Scryer: Netflixs predictive auto scaling engine. `https://medium.com/netflix-techblog/scryer-netflixs-predictive-auto-scaling-engine-a3f8fc922270`, 2013.

[151] Marco A. S. Netto, Carlos Cardonha, Renato L. F. Cunha, and Marcos D. Assuncao. Evaluating auto-scaling strategies for cloud computing environments. In *Proc. of the 2014 IEEE 22Nd International Symposium on Modelling, Analysis & Simulation of Computer and Telecommunication Systems*, MASCOTS '14, pages 187–196, 2014.

[152] Hiep Nguyen, Zhiming Shen, Xiaohui Gu, Sethuraman Subbiah, and John Wilkes. AGILE: Elastic distributed resource scaling for infrastructure-as-a-service. In *Proc. of the 10th International Conference on Autonomic Computing*, ICAC'13, pages 69–82, 2013.

[153] Trond Norbye. Memcached Binary Protocol. `https://https://github.com/memcached/memcached/blob/master/protocol_binary.h`, 2008.

[154] NVM Express Inc. NVM Express: the optimized PCI Express SSD interface. `http://www.nvmexpress.org`, 2015.

[155] Open-iSCSi project. iSCSI tools for Linux. `https://github.com/open-iscsi/open-iscsi`, 2016.

[156] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, and Byung-Gon Chun. Making sense of performance in data analytics frameworks. In *Proc. of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI'15, pages 293–307, 2015.

[157] Jian Ouyang, Shiding Lin, Jiang Song, Zhenyu Hou, Yong Wang, and Yuanzheng Wang. SDF: software-defined flash for web-scale internet storage systems. In *Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 471–484, 2014.

[158] Abhay K. Parekh and Robert G. Gallager. A generalized processor sharing approach to flow control in integrated services networks: The single-node case. *IEEE/ACM Transactions on Networks*, 1(3):344–357, June 1993.

[159] Stan Park and Kai Shen. FIOS: a fair, efficient flash I/O scheduler. In *Proc. of USENIX File and Storage Technologies*, FAST'12, page 13, 2012.

[160] High performance IO Research Group at IBM Research Zurich. Example terasort program. `https://github.com/zrlio/crail-spark-terasort`, 2017.

[161] High performance IO Research Group at IBM Research Zurich. Spark sql benchmarks. `https://github.com/zrlio/sql-benchmarks`, 2017.

[162] Simon Peter, Jialin Li, Irene Zhang, Dan RK Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The operating system is the control plane. *ACM Transactions on Computer Systems (TOCS)*, 33(4):11, 2015.

[163] Adrian Daniel Popescu, Vuk Ercegovac, Andrey Balmin, Miguel Branco, and Anastasia Ailamaki. Same Queries, Different Data: Can we Predict Query Performance? In *Proc. of the 7th International Workshop on Self Managing Database Systems*, 2012.

[164] George Prekas, Mia Primorac, Adam Belay, Christos Kozyrakis, and Edouard Bugnion. Energy proportionality and workload consolidation for latency-critical applications. In *Proc. of the Sixth ACM Symposium on Cloud Computing*, SoCC '15, pages 342–355. ACM, 2015.

[165] Niels Provos and Nick Mathewson. libeventan event notification library. `http://libevent.org`, 2016.

[166] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. Shuffling, fast and slow: Scalable analytics on serverless infrastructure. In *Proc. of the 16th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'19, pages 193–206, 2019.

[167] David Durst Qian Li, James Hong. Thousand island scanner (THIS): Scaling video analysis on AWS lambda. `https://github.com/qianl15/this`, 2018.

[168] Chenhao Qu, Rodrigo N. Calheiros, and Rajkumar Buyya. Auto-scaling web applications in clouds: A taxonomy and survey. *ACM Computing Surveys*, 51(4):73:1–73:33, July 2018.

[169] David Reinsel, John Gantz, and John Rydning. The digitization of the world: From edge to core. `https://www.seagate.com/files/www-content/our-story/trends/files/idc-seagate-dataage-whitepaper.pdf`, Nov 2018.

[170] Francesco Ricci, Lior Rokach, Bracha Shapira, and Paul B. Kantor. *Recommender Systems Handbook*. Springer-Verlag New York, Inc., 1st edition, 2010.

[171] Nilabja Roy, Abhishek Dubey, and Aniruddha Gokhale. Efficient autoscaling in the cloud using predictive models for workload forecasting. In *Proc. of the 2011 IEEE 4th International Conference on Cloud Computing*, CLOUD '11, pages 500–507, 2011.

[172] Ruslan Salakhutdinov and Andriy Mnih. Probabilistic matrix factorization. In *Proc. of the 20th International Conference on Neural Information Processing Systems*, NIPS'07, pages 1257–1264, 2007.

[173] Claude Sammut and Geoffrey I. Webb, editors. *Leave-One-Out Cross-Validation*, pages 600–601. Springer US, 2010.

[174] Samsung. Smart SSD. `https://samsungatfirst.com/smartssd`, 2019.

[175] Samsung Electronics Co. Samsung PM1725 NVMe PCIe SSD. `http://www.samsung.com/semiconductor/global/file/insight/2015/11/pm1725-ProdOverview-2015-0.pdf`, 2015.

[176] R. Sandberg. Design and implementation of the Sun network filesystem. In *In Proc. of USENIX Summer Conference.*, pages 119–130. 1985.

[177] Maheswaran Sathiamoorthy, Megasthenis Asteris, Dimitris Papailiopoulos, Alexandros G. Dimakis, Ramkumar Vadali, Scott Chen, and Dhruba Borthakur. Xoring elephants: novel erasure codes for big data. In *Proc. of the 39th international conference on Very Large Data Bases*, PVLDB'13, pages 325–336, 2013.

[178] Satran, et al. Internet Small Computer Systems Interface (iSCSI). `https://www.ietf.org/rfc/rfc3720.txt`, 2004.

[179] Jörg Schad, Jens Dittrich, and Jorge-Arnulfo Quiané-Ruiz. Runtime measurements in the cloud: Observing, analyzing, and reducing variance. *Proc. VLDB Endow.*, 3(1-2):460–471, September 2010.

[180] Steven S. Seiden. On the online bin packing problem. *J. ACM*, 49(5):640–671, September 2002.

[181] Kai Shen and Stan Park. FlashFQ: A fair queueing I/O scheduler for flash-based SSDs. In *Proc. of USENIX Annual Technical Conference*, ATC'13, pages 67–78. USENIX, 2013.

[182] Zhiming Shen, Sethuraman Subbiah, Xiaohui Gu, and John Wilkes. Cloudscale: Elastic resource scaling for multi-tenant cloud systems. In *Proc. of the 2Nd ACM Symposium on Cloud Computing*, SOCC '11, pages 5:1–5:14, 2011.

[183] Prashant J. Shenoy and Harrick M. Vin. Cello: A disk scheduling framework for next generation operating systems. Technical report, Austin, TX, USA, 1998.

[184] M. Shreedhar and George Varghese. Efficient fair queueing using deficit round robin. In *Proc. of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '95, pages 231–242. ACM, 1995.

[185] David Shue and Michael J. Freedman. From application requests to virtual IOPs: provisioned key-value storage with Libra. In *Proc. of European Conference on Computer Systems*, EuroSys'14, pages 17:1–17:14, 2014.

[186] David Shue, Michael J. Freedman, and Anees Shaikh. Performance isolation and fairness for multi-tenant cloud storage. In *Proc. of USENIX Operating Systems Design and Implementation*, OSDI'12, pages 349–362, 2012.

[187] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop distributed file system. In *Proc. of IEEE Mass Storage Systems and Technologies*, MSST '10, pages 1–10. IEEE Computer Society, 2010.

[188] Solarflare Communications Inc. OpenOnload. `http://www.openonload.org/`, 2013.

[189] Apache Spark. Monitoring and instrumentation. `https://spark.apache.org/docs/latest/monitoring.html`, 2017.

[190] Apache Spark. Spark configuration. `https://spark.apache.org/docs/latest/configuration.html`, 2017.

[191] V. Srinivasan, Brian Bulkowski, Wei-Ling Chu, Sunil Sayyaparaju, Andrew Gooding, Rajkumar Iyer, Ashish Shinde, and Thomas Lopatic. Aerospike: Architecture of a real-time operational DBMS. *Proc. VLDB Endow.*, 9(13):1389–1400, September 2016.

[192] Ioan Stefanovici, Bianca Schroeder, Greg O'Shea, and Eno Thereska. sRoute: Treating the storage stack like a network. In *Proc. of USENIX Conference on File and Storage Technologies*, FAST '16, pages 197–212, Santa Clara, CA, 2016.

[193] Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Transactions on Networks*, 11(1):17–32, February 2003.

[194] Murray Stokely, Amaan Mehrabian, Christoph Albrecht, Francois Labelle, and Arif Merchant. Projecting disk usage based on historical trends in a cloud environment. In *ScienceCloud Proc. of International Workshop on Scientific Cloud Computing*, pages 63–70, 2012.

[195] John D. Strunk, Eno Thereska, Christos Faloutsos, and Gregory R. Ganger. Using utility to provision storage systems. In *6th USENIX Conference on File and Storage Technologies, FAST 2008, February 26-29, 2008, San Jose, CA, USA*, pages 313–328, 2008.

[196] Patrick Stuedi. Crail Storage Performance – Part I: DRAM. `http://crail.incubator.apache.org/blog/2017/08/crail-memory.html`, 2018.

[197] Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, Radu Stoica, Bernard Metzler, Nikolas Ioannou, and Ioannis Koltsidas. Crail: A high-performance i/o architecture for distributed data processing. *IEEE Data Engineering Bulletin*, 40(1):38–49, 2017.

[198] 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 *Proc. of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 182–196. ACM, 2013.

[199] Transaction Processing Performance Council TPC. TPC-DS is a Decision Support Benchmark. `http://www.tpc.org/tpcds/`, 2017.

[200] Animesh Trivedi, Patrick Stuedi, Jonas Pfefferle, Radu Stoica, Bernard Metzler, Ioannis Koltsidas, and Nikolas Ioannou. On the [ir]relevance of network performance for data processing. In *Proc. of the 8th USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'16, pages 126–131, 2016.

[201] Cheng-Chun Tu, Chao-tang Lee, and Tzi-cker Chiueh. Secure I/O device sharing among virtual machines on multiple hosts. In *Proc. of International Symposium on Computer Architecture*, ISCA '13, pages 108–119. ACM, 2013.

[202] Bhuvan Urgaonkar, Giovanni Pacifici, Prashant Shenoy, Mike Spreitzer, and Asser Tantawi. An analytical model for multi-tier internet services and its applications. In *Proc. of the 2005 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '05, pages 291–302, 2005.

[203] Paolo Valente and Fabio Checconi. High throughput disk scheduling with fair bandwidth distribution. *IEEE Transactions on Computers*, 59:1172–1186, 2010.

[204] Shivaram Venkataraman, Zongheng Yang, Michael Franklin, Benjamin Recht, and Ion Stoica. Ernest: Efficient performance prediction for large-scale advanced analytics. In *13th USENIX*

Symposium on Networked Systems Design and Implementation, NSDI'16, pages 363–378, Santa Clara, CA, 2016.

[205] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *Proc. of the European Conference on Computer Systems*, EuroSys'15, Bordeaux, France, 2015.

[206] Jeffrey S. Vitter. Random sampling with a reservoir. *ACM Transactions on Mathematical Software*, 11(1):37–57, March 1985.

[207] Matthew Wachs, Michael Abd-El-Malek, Eno Thereska, and Gregory R. Ganger. Argon: Performance insulation for shared storage servers. In *Proc. of USENIX File and Storage Technologies*, FAST '07, pages 5–5, 2007.

[208] M. Wajahat, A. Gandhi, A. Karve, and A. Kochut. Using machine learning for black-box autoscaling. In *2016 Seventh International Green and Sustainable Computing Conference (IGSC)*, pages 1–8, Nov 2016.

[209] Andrew Wang, Shivaram Venkataraman, Sara Alspaugh, Randy Katz, and Ion Stoica. Cake: Enabling high-level SLOs on shared storage systems. In *Proc. of ACM Symposium on Cloud Computing*, SoCC '12, pages 14:1–14:14. ACM, 2012.

[210] Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 2012.

[211] Jake Wires and Andrew Warfield. Mirador: An active control plane for datacenter storage. In *Proc. of the 15th USENIX Conference on File and Storage Technologies*, FAST'17, pages 213–228, 2017.

[212] Theodore M. Wong, Richard A. Golding, Caixue Lin, and Ralph A. Becker-Szendy. Zygaria: Storage performance as a managed resource. In *Proc. of IEEE Real-Time and Embedded Technology and Applications Symposium*, RTAS '06, pages 125–134. IEEE Computer Society, 2006.

[213] Joel Wu and Scott A. Brandt. The design and implementation of aqua: an adaptive quality of service aware object-based storage device. In *Proc. of the 23rd IEEE / 14th NASA Goddard Conference on Mass Storage Systems and Technologies*, pages 209–218, May 2006.

[214] Neeraja J. Yadwadkar, Bharath Hariharan, Joseph E. Gonzalez, Burton Smith, and Randy H. Katz. Selecting the best VM across multiple public clouds: a data-driven performance modeling approach. In *Proc. of the 2017 Symposium on Cloud Computing*, SOCC'17, pages 452–465, 2017.

[215] C. C. Yeh, J. Zhou, S. A. Chang, X. Y. Lin, Y. Sun, and S. K. Huang. Bigexplorer: A configuration recommendation system for big data platform. In *2016 Conference on Technologies and Applications of Artificial Intelligence (TAAI)*, pages 228–234, Nov 2016.

[216] N. Yigitbasi, T. L. Willke, G. Liao, and D. Epema. Towards machine learning-based autotuning of mapreduce. In *2013 IEEE 21st International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 11–20, Aug 2013.

[217] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proc. of the USENIX Symposium on Networked Systems Design and Implementation*, NSDI'12, pages 15–28, 2012.

[218] Jianyong Zhang, Anand Sivasubramaniam, Qian Wang, Alma Riska, and Erik Riedel. Storage performance virtualization via throughput and latency control. *ACM Transactions on Storage*, 2(3):283–308, August 2006.

[219] Yiying Zhang, Leo Prasath Arulraj, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. De-indirection for flash-based ssds with nameless writes. In *Proc. of USENIX File and Storage Technologies*, FAST'12, page 1, 2012.

[220] Yunqi Zhang, George Prekas, Giovanni Matteo Fumarola, Marcus Fontoura, Inigo Goiri, and Ricardo Bianchini. History-based harvesting of spare cycles and storage in large-scale datacenters. In *Proc. of the 12th USENIX Symposium on Operating Systems Design and Implementation*, OSDI'16, pages 755–770, 2016.

[221] Da Zheng, Disa Mhembere, Randal Burns, Joshua Vogelstein, Carey E. Priebe, and Alexander S. Szalay. Flashgraph: Processing billion-node graphs on an array of commodity SSDs. In *Proc of USENIX Conference on File and Storage Technologies*, FAST '15, pages 45–58, 2015.

[222] Peipei Zhou, Zhenyuan Ruan, Zhenman Fang, Megan Shand, David Roazen, and Jason Cong. Doppio: I/o-aware performance analysis, modeling and optimization for in-memory computing framework. In *International Symposium on Performance Analysis of Systems and Software*, ISPASS'18, 04 2018.

[223] Timothy Zhu, Alexey Tumanov, Michael A. Kozuch, Mor Harchol-Balter, and Gregory R. Ganger. Prioritymeister: Tail latency QoS for shared networked storage. In *Proc. of ACM Symposium on Cloud Computing*, SOCC '14, pages 29:1–29:14. ACM, 2014.