# Flash-based Storage Management in Cloud Computing Datacenter Infrastructures

A Dissertation Presented
by

Zhengyu Yang

to

**The Department of Electrical and Computer Engineering**

in partial fulfillment of the requirements
for the degree of

**Doctor of Philosophy**

in

**Computer Engineering**

**Northeastern University**
**Boston, Massachusetts**

August 2018

*This work is dedicated to my wife, my son, and my parents.*

# Contents

# List of Figures

# List of Tables

# Acknowledgments

I would like to express my sincere gratitude to the following people. It is impossible for me to complete my dissertation without the support of them.

Firstly, I would like to thank my advisor, Prof. Ningfang Mi, for her continuous guidance and encouragement throughout my Ph.D. life with her patience, motivation, enthusiasm, and immense knowledge. I thank her for introducing me to the wonders of scientific research and providing me extensive personal and professional guidance. I could not have imagined having a better advisor and mentor for my Ph.D. study.

Secondly, I would like to offer my special thanks my Ph.D. defense committee members, Prof. Yunsi Fei and Prof. Straitis Ioannidis, for their great support and insightful feedback and comments on my dissertation.

Thirdly, my thanks also go to my labmates Ms. Janki Bhimani, Mr. Danlin Jia, Mr. Han Gao, Dr. Jianzhe Tai, Dr. Yi Yao, Mr. Cho-Hsien Lin and Ms. Mahsa Bayati for working together, and for all the fun we have had in the past several years.

Fourthly, I am grateful to all of my internship mentors, Prof. Manu Awasthi, Dr. Mrinmoy Ghosh, Mr. Vijay Balakrishnan, Mr. David Evans, and Mr. Ping Wong, who guided me during my three summer internships in the Memory Solutions Lab of Samsung Semiconductor Inc.

In addition, I wish to acknowledge the help provided by my research collaborators: Prof. Bo Sheng, Prof. Chiu C. Tan, Prof. Miriam Leeser, Prof. Steven Swanson, Prof. Jiayin Wang, Prof. Ying Mao, Dr. Xiaoyun Zhu, Dr. Jack Lo, Dr. Jingpei Yang, Dr. Changho Choi, Dr. Qiumin Xu, Mr. Morteza Hoseinzadeh, Mr. Allen Andrews, Mr. Clay Mayers, Mr. Rory Bolt, Mr. John Artoux, Mr. Yufeng Wang, Ms. Teng Wang, Mr. Nam Nguyen, Mr. Rajinikanth Pandurangan, Mr. Jake Roemer, Mr. Mark Groman, Mr. Deng Liu, and Mr. N.H.V. Krishna Giri.

Finally, nobody has been more important to me in the pursuit of my Ph.D. degree than the members of my family. I am grateful to my wife Eri, my son Patrick, and my parents Jenny and James, for always supporting me, loving me, and standing by me through the tough times and bringing happiness into my life.

# Abstract of the Dissertation

Flash-based Storage Management in Cloud Computing Datacenter Infrastructures

by

Zhengyu Yang

Doctor of Philosophy in Electrical and Computer Engineering

Northeastern University, August 2018

Dr. Ningfang Mi, Advisor

A basic credendum of cloud computing can be summarized as: user devices are *light* terminals to assign jobs and gather results, while those *heavy* computations are conducted on remote distributed server clusters. This *light-terminal-heavy-server* structure makes high availability no longer an option, but a requirement in today's datacenters. Furthermore, when bringing compute and storage capabilities into balance, we find that the biggest challenge here is closing the gap between compute and storage performance to shift storage's curve back towards Moore's law [1]. In detail, the time consumed to wait for I/Os is the main cause of idling and wasting CPU resources, since a large number of popular cloud applications are I/O intensive, such as video streaming, file sync and backup, and data iteration for machine learning [2]. Thus, storage I/O is the biggest bottleneck in large-scale datacenters. To address this bottleneck, Solid State Drives (SSDs) are widely being deployed as a per-virtual disk, second-level cache of Hard Disk Drives (HDDs) in SSD-HDD hybrid storage systems to improve I/O access performance, ascribing to SSD's high I/O throughput, low I/O latency and low power consumption. Recently, the capital expenditure of flash-based SSDs keeps decreasing and the storage capacity of SSDs keeps increasing. Consequently, the "sales pitch" of traditional HDDs as backend storage – low cost and large capacity – is no longer unique, and eventually, these HDDs will be replaced by low-end SSDs. As a result, it is widely believed that all-flash multi-tier storage systems will be adopted in the enterprise datacenters running big data platforms in the near future. Therefore, in this dissertation, we focus on studying storage resource management for both SSD-HDD hybrid and all-flash storage systems, and investigating storage optimization for big data platforms. We aim to achieve high availability of storage capacity and accessibility by improving both performance and reliability in these storage systems.

# Chapter 1

# Introduction

*"Data creation is exploding. 92% of the world's data was created in the last two years alone. At the current rate, the world's data storage capacity will be overtaken by next spring. It will be nothing short of a catastrophe." — Silicon Valley, Sand Hill Shuffle* [3, 4].

In the era of cloud computing, one of the most significant research areas is the data storage infrastructure for large-scale data processing platform. According to an IBM's report [5], 2.5 quintillion bytes of data created from the user side are stored in the cloud every day. Google also reported that "the web is growing by millions of pages per day, which increases the cost of building and serving a web index" [6]. Visa even spent almost a month to interrogate 73 billion transactions [7]. Huge capacity is highly demanded to store the enormous amount of data. Besides, user applications also request their files to be available for lightning-fast download and be accessible from any devices. Meanwhile, cloud vendors strive to optimize the usage of their limited and oversold resources to serve their clients with the consideration of their different workload patterns and Service Level Agreements (SLAs). Moreover, cloud vendors have the concern on their system's reliability for disaster and failure recovery, and need to minimize the Total Cost of Ownership (TCO) for purchasing and maintaining their system due to device endurance. Thus, large capacity, high performance and reliability, and low maintenance cost are no longer options, but necessary requirements in today's datacenter ecosystems.

## 1.1 SSD-HDD Hybrid Storage Systems

A common approach to solving the storage I/O bottleneck is to parallel I/Os to multiple Hard Disk Drives (HDDs) in the Redundant Array of Independent Disks (RAIDs) mode [8]. However,

HDDs still need to move their heads over the surface of spinning platters to read or write the data, thus the performance improvement from RAID is limited. Motivated by this, lots of big data processing applications strive to store intermediate data to memory as much as possible, such as Apache Spark [9]. However, relying on memory is too costly, and memory's capacity is far more limited than HDD's (e.g., less than 1TB per server in 2018). As a result, memory alone is not able to support large-scale cloud computing use cases. Fortunately, a new type of storage device without moving parts – Solid State Drive (SSD) – was invented to achieve spatial capacity and acceptable price. Ascribe to this, since 2008, as a "sweet spot" between Random-Access Memory (RAM) and HDDs, SSDs started to be adopted in the server market to break the asymmetric read/write IOPS (I/O per second) barrier dramatically, and became one of the promising solutions to speedup storage systems as a cache or a fast tier for slow HDDs [10]. On the other hand, we also need to face the fact that compared to HDDs, SSDs still have the limitation of a relatively small spatial capacity and high costs per GB. As a result, it is important to let multiple Virtual Machines (VMs) efficiently share the SSD resources in an SSD-HDD hybrid storage system.

In most of the exiting shared virtualization platforms, SSD is statically pre-allocated to each Virtual Machine Disk file (VMDK) for simplicity, and the caching algorithm decides the cache admission and eviction for each VM only based on I/O requests from that particular VM regardless of I/Os from the others. It is difficult for the hypervisor to cost-effectively partition and allocate SSD resources among multiple VMs, particularly under diverse I/O demands, because it lacks a global view of the cluster-wide disk I/O activities. Therefore, we first focus on addressing a critical design problem for a virtualized storage system – "*how to dynamically partition flash-based SSDs among multiple VMs and cost-effectively update the content of SSDs according to VM workload changes?*". We propose a Global SSD Resource Management solution, named "GREM", to fully leverage the outstanding performance of shared SSD resources under the global view of caching management. In detail, GREM takes dynamic I/O demands of all VMs into consideration to split the entire SSD space into a long-term zone and a short-term zone and update the content of SSDs in these two zones cost-effectively. Intuitively, the long-term zone is designed for reserving SSD resources for each VM, in order to cache their own hottest (or the most popular) data without any pollution from other VMs. Such a long-term zone is expected to guarantee high hit ratios from VMs that have cache-friendly workloads. On the other side, the short-term zone is used to absorb and handle bursty I/Os (mostly from VMs with cache-unfriendly workloads) by being fairly competed among VMs according to their data popularities. We use a coarse temporal (e.g., 5 $min$) and spatial (e.g., $1MB$) granularity to update the contents of SSDs in the two zones for reducing the cost of managing and operating

SSD resources. In addition, there are two follow-up questions in the design of this new global SSD manager, i.e., "*how to dynamically partition SSD resources into two zones?*" and "*how to further dynamically allocate SSD resources in the long-term zone to different VMs?*". These questions are challenging because VMs with heterogeneous I/O workloads (e.g., with a mix of cache-friendly and cache-unfriendly workloads) are sharing SSD resources, and their I/O access patterns can frequently change across time. Thus, GREM online monitors the changes in I/O demands of all VMs as well as the SSD allocation performance (e.g., I/O hit ratios) and uses this information to further dynamically partition SSD resources between two zones, and reserve different amounts of SSD resources to each VM.

Besides I/O performance, another common concern in such a hybrid datacenter is reliability. The ability to reliably store data is an essential requirement for millions of people and virtually all industries. A common technique to address this need is replication, which however brings redundancy overheads for consistency maintenance, such as network traffic, I/O bandwidth, and storage space [11, 12]. In order to balance replication and performance, we propose an automated distributed replication manager, called "AutoReplica", to address concerns over data center cluster disaster recovery and data reliability. AutoReplica utilizes the SSDs of remote neighbor server nodes to replicate hot data cached in local SSD tier of each server node, since comparing to HDD, SSD is relatively not a "safe destination" although it can preserve the data after power off [13]. AutoReplica also automatically balances loads among server nodes and conducts seamless online migration operations by using a novel "migrate-on-write" scheme. Additionally, AutoReplica enhances the overall performance by conducting parallel I/O prefetching from both main node and its available replica nodes when the system is under the idle I/O traffic status.

## 1.2 All-Flash Storage Systems

As time goes by, SSD-HDD hybrid solutions are no longer competent to meet the current big data processing requirements, due to the huge I/O speed gap between SSDs and HDDs [14]. On the other hand, the capital expenditure of flash-based SSDs keeps decreasing and the storage capacity of SSDs keeps increasing. As a result, the "sales pitch" of traditional spinning hard disk drives as backend storage, i.e., low cost and large capacity, is no longer as unique as before, and eventually, these HDDs will be replaced by low-end SSDs which have large capacity but perform orders of magnitude better than HDDs. Thus, it is widely believed that the SSD-HDD solution is just for a transition period, and all-flash multi-tier storage systems will be adopted in the enterprise

datacenter in the near future, similar to what happened to HDD-tape hybrid storage solution 30 years ago [15]. For example, high-end Non-Volatile Memory Express (NVMe) SSDs can replace Serial Advanced Technology Attachment (SATA) SSD, and low-end Triple-Level Cell (TLC) SSDs will replace HDDs.

While all-flash datacenters will revolutionarily change the ecosystem of both disk manufactures and cloud vendors, we observed that current caching and tiering solutions for hybrid storage systems cannot fully utilize each SSD tier in the all-flash system due to the following three reasons: (1) all-flash storage systems do not have a large speed difference (e.g., 10x) among each tier; (2) SSDs are expensive, and it is costly to maintain duplicated copies in two SSD tiers (e.g., one data copy in the cache tier and one data copy in the backend tier if write-back or write-through policy is used); and (3) different specialties of each tier (such as high performance, high capacity, etc.) should be taken into consideration. Motivated by these new requirements, we develop an automatic data placement manager, called "AutoTiering", to handle VMDK allocation and migration in an all-flash multi-tier datacenter with the consideration of characteristics of workload pattern and specifications of each SSD tier. By this way, AutoTiering is able to better utilize the storage resource, optimize the performance, and reduce the migration overhead. In particular, AutoTiering first predicts VM's performance change on different tiers with different specialties without conducting real migration through a micro-benchmark-based sensitivity calibration and regression session. It then associates VMDKs with an appropriate SSD tier during runtime since different VMs may have different benefits of being upgraded to a high-end tier.

Nevertheless, the industry is very sensitive to the cost of owning the all-flash datacenter, including purchasing and maintaining SSDs. SSDs are still very expensive since they have limited write cycles and also suffer from write amplification. Write amplification is caused by a number of factors specific to flash devices including erase-before-rewrite, background garbage collection, and wear leveling [16]. Therefore, we develop an I/O workload dispatcher, called "MINTCO", to balance endurance based on write amplification estimation for different workload patterns, and minimize the total cost of ownership by deploying and allocating applications to a shared all-flash storage system based on our new TCO model. In detail, to accurately reflect the SSD-intensive storage datacenters, the TCO model we proposed considers multiple factors, like SSD lifetime, workload sequentiality, device wear degree. Moreover, we conduct real experiments to measure and characterize write amplification under different workloads, and reveal the relationship between write amplification and workload sequential ratio.

## 1.3 Storage Systems for Big Data Platforms

Apart from boosting up the storage system performance and reliability from a hardware perspective (i.e., utilizing SSDs), we also need to put efforts on the software level of the storage infrastructure of big data platforms. We observe that the limitation induced by software overheads prevents users to perfectly perceive this performance advancement. In fact, the overhead of the legacy kernel I/O stack, which has been optimized for slow HDDs, becomes more noticeable as the storage devices and the connection interfaces get faster. For example, VMware ESXi [17] is one of the most common commercial virtualization platforms which can potentially use the NVMe SSDs in the most efficient way by leveraging the massive parallelism and isolation characteristics of NVMe SSDs. However, VMs access the storage resource through a single submission and completion queue in the NVMe driver, regardless of high levels of parallelism provided by NVMe. This inefficiency originates from the NVMe driver in the hypervisor and has become a bottleneck in the storage I/O stack. Thus, we propose H-NVMe, a hybrid NVMe framework on VMware ESXi, to best utilize NVMe SSDs in virtualization platforms. Specifically, H-NVMe provides two different working modes, "*Parallel Queue Mode*" and "*Direct Access Mode*". In the former working mode, H-NVMe circumvents the built-in *Adapter Queue* of ESXi by emptying it and spreading out its entities among multiple lightweight subqueues in our customized NVMe driver more efficiently to use the parallelism of storage devices. The latter working mode bypasses all the hypervisor queues and directly connects the trusted user application threads to the NVMe *Driver Queue*, to achieve better performance isolation. H-NVMe can work in either of these two modes either in whole or in partial.

Meanwhile, with the rise of big data analytics and cloud computing, cluster-based large-scale data processing has become a common paradigm in many applications and services. Online companies of diverse sizes, ranging from technology giants to smaller startups, routinely store and process data generated by their users and applications in the cloud. Data-parallel computing frameworks, such as Apache Hadoop [18] and Spark [19, 9], are employed to perform such data processing at scale. Jobs executed over such frameworks comprise hundreds or thousands of identical parallel subtasks, operating over massive datasets, and executed concurrently in a cluster environment. The time and resources necessary to process such massive jobs are immense. Nevertheless, jobs executed in such distributed environments often have significant computational overlaps: different jobs processing the same data may involve common intermediate computations. Such computational overlaps arise naturally in practice. Exploiting such computational overlaps has a tremendous potential to drastically reduce job computation costs and lead to significant performance improvements.

Existing data-parallel computing frameworks, such as Spark, incorporate caching capabilities in their framework in a non-automated fashion. The decision of which computation results to cache rests on the developer that submits jobs: the developer explicitly states which results are to be cached, while cache eviction is implemented with the simple policy (e.g., LRU or FIFO); neither caching decisions nor evictions are part of an optimized design. Crucially, determining which outcomes to cache is a hard problem when dealing with jobs that consist of operations with complex dependencies. Motivated by them, we develop an adaptive algorithm for caching in a massively distributed data-parallel cluster computing environment, handling complex and massive data flows. Specifically, a mathematical model is proposed for determining caching decisions that minimize total work, i.e., the total computation cost of a job.

## 1.4 Dissertation Contributions

As illustrated in Fig. 1.1, in this dissertation, the main contributions regarding flash-based storage resource management are summarized as follows:

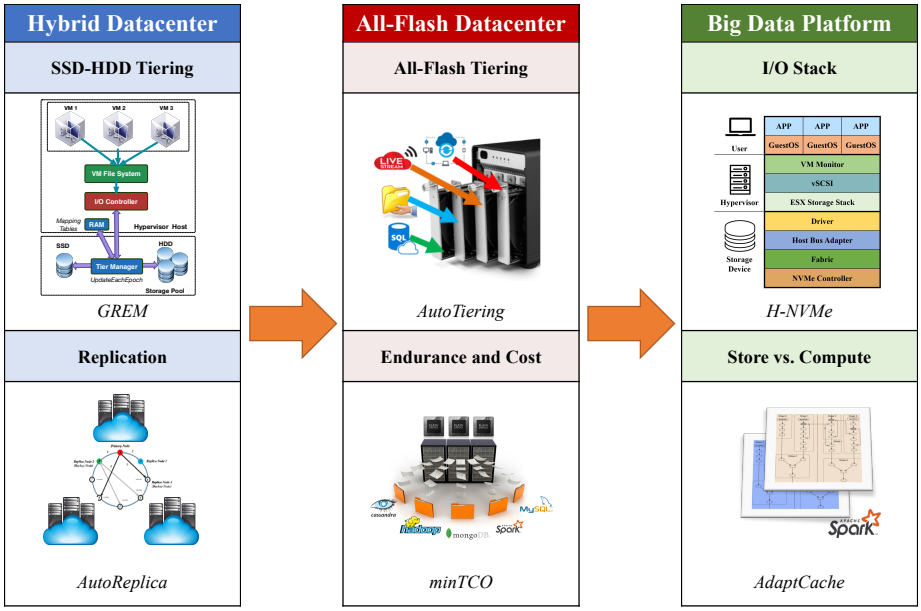



Figure 1.1: Components of this dissertation.

1. A flash resource manager (GREM [20]) to dynamically allocate flash resources among multiple clients (or VMs), especially when their workloads change across time.

2. A data replication manager (AutoReplica [21, 22]) to address the reliability and accessibility requirements by automatically selecting remote SSDs from neighbor node(s) to backup local SSD cached data.

3. An automatic data placement manager (AutoTiering [23]) to handle VM allocation and migration in an all-flash multi-tier datacenter based on VMware's ESXi.

4. An online workload allocation algorithm (MINTCO [24]) for all-flash datacenter storage systems, to minimize the TCO with the consideration of capital and operational costs, and the estimated lifetime of SSD devices under different workloads, resource restrictions and SLAs.

5. A hybrid NVMe deployment framework ("H-NVMe" [25]) to improve the I/O stack of a virtualized storage system for better utilizing the high parallelism of current cutting-edge NVMe SSD devices.

6. A cache scheme (AdaptCache [26]) for modern multi-stage big data platforms (such as Apache Spark) with the consideration of each intermediate dataset's arrival and reuse rate, and re-computing and fetch cost.

## 1.5 Dissertation Organization

The remainder of the dissertation is organized as follows. Chapter 2 provides a brief overview of the background and related work of this dissertation. Chapter 3 focuses on SSD-HDD hybrid storage systems. Sec. 3.1 introduces our novel SSD resource management scheme "GREM" to allocate a suitable amount of SSDs to heterogeneous VMs. Sec. 3.2 then presents our replication manager in SSD-HDD hybrid storage clusters. In Chapter 4, we discuss resource management schemes in all-flash datacenter storage systems. Sec. 4.1 describes our automatic data placement manager "AutoTiering" which handles VMDK allocation and migration in an all-flash multi-tier datacenter. Sec. 4.2 introduces our online workload allocation algorithm "MINTCO" which strives to reduce the TCO of all-flash datacenter with the consideration of flash endurance and write amplification. In Chapter 5, we discuss storage resource optimization in big data platforms. Sec. 5.1 proposes a hybrid framework of NVMe-based storage system called "H-NVMe" to simplify the I/O stack in virtual machine hypervisor. Sec. 5.2 shows an adaptive algorithm to automates the decision of cache or re-compute intermediate datasets in Apache Spark platform. We finally draw the conclusion and present our future work in Chapter 6.

# Chapter 2

# Background and Related Work

Substantial work has been done to improve I/O operations, reliability, and modeling total cost of ownership in datacenters at both hardware and software levels. In this chapter, we discuss the background knowledge as well as the evolutionary inclination towards our work.

## 2.1 Storage Technology Overview

The storage technology is the means of maintaining digital data in form of binary information, and plays a key role in the latency of a device. In this subsection, we discuss the evolution of storage device, NVMe SSD, write amplification and endurance.

### 2.1.1 Storage Device Evolution

In HDD devices, data is stored in form of sequential changes in the direction of magnetization in a thin film of ferromagnetic material on a disk. Reading and writing to such a device has mechanical nature of the rotating disks and moving heads [27]. Later on, flash memory cells came into play, and a new trend of storage technology formed by introducing flash-based SSDs. Since flash memory cells need to be erased before storing new data, a Flash Translation Layer (FTL) makes SSDs to present HDD-like I/O interface to an operating system. Even though the latency of an SSD varies under different workloads because of FTL, it is orders of magnitude faster than HDDs. As a result, the new trend in developing SSD devices is rapidly going on to provide cheap storage devices with large storage capacity. There have been a large number of studies on other Storage Class Memory (SCM) including Phase-Change Memory (PCM) [28], Resistive RAM [29], and 3D XPoint [30], which use these storage devices either as primary storage in form of Non-Volatile Dual In-line Memory

Module (NVDIMM) [31, 32], or as secondary storage in form of SSDs [33, 34]. The ultimate goal of developing SCM is to provide near-DRAM latency and also to persistently maintain data. Until 2017, no matured SCM-based devices appeared in the market. In fact, flash-based SSDs is replacing HDDs at unprecedented speeds. Such modern SSDs tend to exploit the parallelism of multiple flash chip, and reorder I/O requests for better scheduling [35], in order to amortize per-request latency overhead and achieve high bandwidth by performing concurrent data transfers.

On the other hand, with the emergence of new and fast storage technologies, hardware I/O bus interface also has experienced significant upgrades to facilitate the use of these modern technologies. Parallel ATA interface (evolved from Integrated Drive Electronics, i.e., IDE) has become outdated as it was standardized for slow HDDs with a few megabytes of data transfer per second. Recent fast storage devices usually support Serial ATA (SATA) with a few gigabytes per second transfer rates. However, new storage devices demand much higher transfer rates. The maximum I/O bandwidth is determined by the I/O bus interface which is migrating from IDE and SATA toward Peripheral Component Interconnect Express (PCIe) and is expected to support up to 128 GB/s by 2019 in PCIe 5.0 [36].

### 2.1.2 Non-Volatile Memory Express SSD

Since the 1990s, flash-based memory and SSDs have inherently strong points compared to a traditional hard disk. Recently, flash-based SSDs have been widely adopted in datacenter storage systems because of their several sales pitches, including fast access speed, non-volatility, shock resistance, and low power consumption. In terms of connection interface, in the past, SATA, Serial Attached SCSI (SAS) or Fibre Channel (FC) buses were most common interfaces for SSDs. In fact, according to the enterprise storage system marketplace, SATA is still the most typical way for interfacing SSDs and Operating Systems (OSs). Also, there are a number of PCIe-based bus interfaces for high-end SSDs. But there is no standard specification interfaces. After standardizing a universal interface for SSDs, the OS is able to use only one driver to interact with all SSDs. Thus, it is no longer needed to use additional resources to develop specific interface drivers. Around 2011, NVMe [37] is released as a scalable host controller interface designed for both enterprise and client systems to use SSDs over PCIe. Recent NVM-Express standard [38] abridges the I/O path with several deeper queues. It provides a device driver to bypass the block layer and the Small Computer System Interface (SCSI) I/O subsystem. This driver directly issues the requests to a deeper hardware queue (up to 64K in depth) which enervates the need for background queue running

context in the I/O completion path. Consequently, host hardware and software can fully exploit the highest level of parallelism offered by modern SSDs which form a single unit that plugs into the PCIe bus. In this regard, a new log-structured file system, called "NOVA" [39], is proposed to maximize performance on hybrid memory systems while providing strong consistency guarantees. Studies [40, 41] characterize the performance of persistent storage option (through data volume) for I/O intensive as well as dockerized applications for NVMe SSDs. Study [42] further conducts performance analysis of non-volatile memory host controller interface in high-performance I/O systems.

### 2.1.3 SSD Write Amplification and Endurance

Flash devices have a unique property that they cannot be re-written unless they have been erased. Also, the minimum granularity of an erase operation is in the order of MBs (e.g., blocks), while the granularity of writes is much smaller, in the order of KBs (e.g., pages). Meanwhile, flash devices have limited write life cycles. Thus, for the purpose of wear-leveling, the logical address space in flash devices is dynamically mapped to the physical space and the mapping changes with every write [43]. Specifically, flash devices have a software called FTL running on them to manage the erase before re-write and wear-leveling requirements, as shown in Fig. 2.1. The FTLs have to schedule periodic garbage collection events to de-fragment their write data. These garbage collection events can lead to extra writes that have not been generated by the host. Additionally, SSD reserves a user-invisible space (i.e., over-provision), which is helpful to reduce the write amplification during these above-mentioned events to some extent. However, since flash devices have limited write-erase cycles, the mismatch between the two types (logical and physical) of writes can still cause the SSD to fail much more quickly than expected. All of these above-mentioned factors cause the write amplification in SSD, which is an undesirable phenomenon where the actual amount of data written to the device is larger than the logical amount of data written by a workload. To measure the write amplification degree caused by these facts, a commonly used metric is Write Amplification Factor ("$WAF$", henceforth referred to as "$A$" and "$WA$"). We define $A$ as the ratio between the total physical write data written by the SSD and the total logical data written by the workload: $A = \frac{W_P}{W_L}$, where $W_L$ denotes the logical write amount (in bytes) and $W_P$ denotes the physical, device-level I/O writes as seen by the SSD (also in bytes). Large values of $A$ lead to increase I/O latency, shorten the SSD's working lifetime, and increase power consumption.

A lot of research studies have been done to model the WAF function. For example,

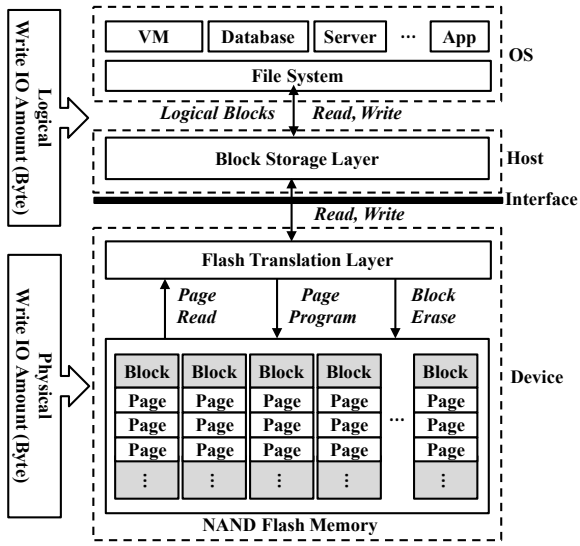



Figure 2.1: An example of I/O path from OS to device.

analytical models [44, 45] reveal the relationship between workload characteristics and different garbage collection policies (i.e., cleaning algorithms). However, these models have ignored a factor, sequential ratio, which is becoming increasingly important, especially in the NoSQL database community. With the proliferation of log structured merge tree (LSM tree) based NoSQL databases, there is a lot of uptick in the amount of sequential traffic being sent to SSDs [46]. LSM-tree-based databases capture all the writes into a large in-memory buffer. When the buffer is full, all cached data will be flushed to disk as a large, multi-gigabyte sequential write [47]. Another similar case is about write-intensive workloads that execute within virtual machines and Docker containers, where most of the write traffic to SSD is usually sent out as large, sequential writes [48, 49, 50]. Hence, it is becoming increasingly essential to understand the WAF, performance of SSDs, device wornout, and most importantly, the total owning cost of datacenters from a workload-centric view.

## 2.2 I/O Management Technology Overview

Historically, a common solution to improve HDD's I/O throughput is expanding parallelism. RAID is a matured methodology to this end, which combines multiple physical disk drive components into a single logical unit and distributes data across disk drives. However, because of the mechanical binding essence of HDDs, the improvement brought by parallel I/Os is still limited. Another solution is to associate a memory space (i.e., DRAM) to each application of the server host for caching hot

data, and leave cold data to the back-end storage pool. However, this solution is not affordable in the big data era, because DRAM is capacity-limited, volatile, and expensive while there are a huge amount of I/Os and spatial capacity requests generated by cloud applications. Simply relying on HDD or DRAM cannot solve the problem, so we discuss some substantial flash-based caching and tiering solutions aiming to improve I/O operations in the following subsections.

### 2.2.1 Caching and Tiering Solutions in Hybrid Systems

SSD can be used either as a cache for HDD or as a distinct storage tier. The main difference is that the cache approach has two copies of the hot data, one in the SSD and one in the HDD (two copies are synced under the *write through* policy, and are not synced under the *write back* policy). While, the tiering approach simply migrates data between tiers and only keeps one version of the dataset. Host-side caches are being widely accepted in modern storage systems. ARC [51] and LRU-K [52] are commonly used caching algorithms that consider the frequency and recency of workloads. Inspired by ARC and based on Clock [53], CAR and CART [54] are developed to inherit all advantages of ARC virtually, but not serialize cache hits behind a single global lock. [55] uses SSD as a disk cache and adopts the wear-level aware replacement policy based on LRU. SieveStore [56] is a selective and ensemble-level disk cache by using SSDs to save the popular dataset. A hybrid storage system, called "Hystor" [57], is developed to fit SSDs into the storage hierarchy. Hystor identifies the performance- and semantically-critical data, and retains these data to the SSD tier. Argon [58] partitions the memory cache among different services, providing performance isolation with respect to the hit of each service. The difference between Argon and our dissertation is that Argon aims to optimize the cache hit rate for individual service, while we strive to optimize both the overall cache utilization and I/O hit ratio. Jigsaw [59] is a CPU cache partitioning solution, which mainly focuses on resource constrains (i.e., compute and storage). mClock [60] follows the proportional-share fairness approach which subjects to minimize reservations and maximize limits on the I/O allocations for VMs. S-Cave [61] is a hypervisor-based optimization design which is based on runtime working set identification, while we are focusing on exploring a different dimension by monitoring changes in data locality, burstiness, and I/O popularity. Recently, a new allocation model based on the notion of per-device bottleneck sets is introduced in [2] which can balance different types of resources through an optimization framework. vCacheShare [62] presents a dynamic, self-adaptive framework for automated server flash cache space allocation in virtualization environments. However, vCacheShare treats SSD as a read-only cache and bypasses write I/Os to the disk, which unfortunately will degrade

the overall hit ratio.

### 2.2.2 Data Placement in Multi-Tier All-Flash Systems

SSD-HDD based solutions may work for a limited number of users (VMs) with mediate I/O intensity and small working set size. But, for the era of super-scale clusters (e.g., cloud computing, IoT in 5G network), the I/O bottleneck gets mitigated, but not resolved [56]. The main reason is that in both SSD-HDD caching and tiering approaches, there still exists a huge performance gap between SSDs and HDDs. With the decreasing price and increasing capacity of SSDs, a promising solution to quench this gap is to setup an all-flash datacenter which is becoming a reasonable solution in the near future. We summarize the specs of SSDs with different ends available in the market by July 2017 in Table 2.1. As we can see, an all-flash multi-tier solution can be built from different SSDs with different specialties, e.g., *super-performance tier* with 3D XPoint SSD [30], *high-performance tier* with NVMe and SLC SSDs, and *large-capacity tier* with MLC and TLC SSDs [63].

| **SSD Type** | **Cost ($/GB)** | **Max Size (Bytes)** | **Read Time ($\mu s$)** | **Write Time ($\mu s$)** |
|---|---|---|---|---|
| 3D XPoint | 4.50 | 375G | 10 | 10 |
| NVMe | 0.57 | 3.2T | 20 | 20 |
| SLC SATA3 | 0.64 | 480G | 25 | 250 |
| MLC SATA3 | 0.30 | 2T | 50 | 750 |
| TLC SATA3 | 0.28 | 3.84T | 75 | 1125 |

Table 2.1: Performance and cost of different SSDs in July 2017.

There are a few studies focused on tiering solutions in all-flash storage systems. For example, [34] presents a study of caching and tiering approaches by evaluating phase change memory for enterprise storage systems. With the aim of further reducing the overall capacity in a multi-tier storage system, Gmacha et al. [64] propose a resource management solution to periodically re-compute VM assignments. [65] introduces mathematical model formulations for big data application performance and VMs migration among tiers in order to minimize the overhead of data migration. Recently, Petersen et al. develop a hybrid flash array for HPC storage systems [66], which only buffers small I/Os while allowing large sequential I/Os to access the flash devices directly.

### 2.2.3 I/O Stack in VM Hypervisors

As the hardware latency constantly decreases, many studies have been conducted to diminish the storage stack latency, along the way. There are a tremendous number of studies that strive to reduce the kernel overhead by eliminating unnecessary context processing [67, 68, 69], employing a polling mechanism instead of interrupts [70, 71, 67, 69], and performance isolation [72, 73]. Shin et al. [67] present a low-level hardware abstraction layer interface which curtails scheduling delays caused by extra contexts to optimize the I/O path. Yu et al. [68] demonstrate six optimization schemes to fully utilize high performance introduced by fast storage devices. The proposed schemes in [68] relies on a hardware support to expand parallelism inside SSDs. Similar to [67], they also eliminate context switches in the I/O path. Additionally, they exploit polling I/O completion, merging I/O, and double buffering. Likewise, Yang et al. [70] compare polling-based and interrupt-based I/O paths, and eventually come up with the fact that synchronously polling model for I/O completion is much faster in the era of non-volatile memories and fast SSDs. Multistream is also a new technology to separate I/O streams based on their characteristics to fully utilize those NVMe queues in the I/O path [74, 75] .

To maximize parallelism further, P. Kumar and H. Huang [76] propose Falcon which is a single flush thread per drive, instead of per volume, and separates I/O batching and I/O serving in the storage stack. EAD [77] investigates a data deduplication technique to improve the performance of modern datacenters. Similarly, M. Lee et al. [78] propose to isolate read and write requests through separate I/O queues with the aim of eliminating write interference on read performance in NVMe SSD. Besides the above design towards optimizing the I/O path, the kernel also imposes overhead to the storage stack. To address this issue, researchers have suggested to grant direct access to the user applications without involving the kernel. Caulfield et al. [71] present a new hardware/software architecture which achieves high performance by skipping the kernel involvement and leaving the file-system permission checking phase to the hardware (i.e., their special storage device, Moneta [79]). Aerie [80], another flexible file-system, is introduced by Volos et al. to expose the storage devices to user applications for accessing without kernel interaction. Kim et al. propose NVMeDirect [69] which is a user I/O framework allowing user applications to directly access commercial NVMe SSDs by associating NVMe I/O queues to them upon request.

### 2.2.4 I/O Workload Analysis

Effective workload studies can imply the accurate modeling, simulation, development, and implementation of storage systems. For example, SRCMap [81] presents an energy proportional storage system by effectively characterizing the nature of I/O access on servers using workloads from three production systems. Study [82] introduces 12 sets of long-term storage traces from various Microsoft production servers, and analyzes trace characterizations in terms of block-level statistics, multi-parameter distributions, file access frequencies, and other more complex analyses. FIU [83] captures I/O workload traces from actively-used production storage systems, all of which reveal surprisingly high levels of content similarity for both stored and accessed data. In Study [84], the authors analyze the I/O workloads captured over 284 billion requests from five different Facebook Memcached use cases over several days. They focus on request data composition rate, cache efficacy, temporal patterns, and application use cases. Study [85] uses data traces obtained from a real data center to develop such capabilities, and introduces a method based on Hidden Markov Modeling to characterize the temporal correlations in the discovered VM clusters and to predict variations of workload patterns. Study [86] proposes a distributed VM I/O performance measurement and analysis framework.

### 2.2.5 Memory Management in Big Data Platforms

In the era of big data, it becomes highly demanded to analyze and process a large amount of data in a short time period. To meet this requirement, two types of in-memory processing systems are developed by researchers [87]. The first type is data analytics systems focusing on batch processing such as Spark [9], Hadoop [18], SINGA [88], Giraph [89], and GridGain [90]. The second type is real-time data processing systems such as Storm [91], Spark Streaming [92], MapReduce Online [93]. Meanwhile, memory management is also a well-studied topic across these in-memory processing systems. Memcached [94] and Redis [95] are highly available distributed key-value stores. Megastore [96] offers a distributed storage system with strong consistency guarantees and high availability for interactive online applications. MemTune [97] is a dynamic memory manager based on workload memory demand and in-memory data cache needs. A number of studies have also been done for modeling the multi-stage frameworks. Study [98] compares the performance in both time and memory cost between Hadoop and Spark, and concludes that Spark is, in general, faster than Hadoop in iterative operations but Spark has to pay for more memory consumption. Study [99] presents an efficient application-aware storage system for Hadoop platform with heterogeneous

clusters. Study [100] proposes a simulation driven prediction model that can predict job performance with high accuracy for Spark. A novel analytical model is designed in [101], which can estimate the effect of interference among multiple Spark jobs running concurrently on job execution time. There are some heuristic approaches to evict intermediate data in big data platforms. Least Cost Strategy (LCS) [102] evicts the data which lead to minimum recovery cost in future. Least Reference Count (LRC) [103] evicts a cached data block that has the smallest reference count, where the reference count records the number of dependent child blocks that have not been computed yet. Weight Replacement (WR) [104] is another heuristic approach to consider computation cost, dependency, and sizes of intermediate datasets. ASRW [105] uses intermediate data reference values to improve the memory cache resource utilization rate and the running efficiency of applications. Meanwhile, researchers also focus on the computing and storing costs problem in online services. Study [106] develops cost metrics to compare storage vs. compute costs and further suggests when to use a transcoding on-the-fly solution can be more cost-effective. Weighted-Rank Cache replacement Policy (WRCP) [107] uses parameters such as access frequency, aging, and mean access gap ratio to calculate the cost of retrieval. Although, these heuristic approaches strive to use optimization frameworks to solve the problem, they only focus on a single job, and ignore cross-job intermediate dataset reuse.

## 2.3 Reliability Technology Overview

Besides performance, the need for reliable infrastructure to store the information necessitates the continual advancement of information technology solutions addressing data storage system performance and reliability. In the following subsections, we discuss the background knowledge of datacenter reliability.

### 2.3.1 Replication Management

Replicas are useful in an incident of data loss for recovery [108]. Data loss can be incurred by many factors such as software-hardware failure, natural disasters at data center location [109], power surge, etc. Moreover, when there are more highly-visited files gathered in some nodes with poor storage capacity, it will cause a hot issue which may reduce the overall performance of the system. As a result, replication is used to guarantee performance in such a critical situation. Facebook's implementation [12] based on Apache Hadoop Distributed File System (HDFS) constrains the

placement of replicas to smaller groups in order to protect against concurrent failures. MongoDB [11] is a NoSQL database system that uses replicas to protect data. Its recovery scheme is based on the election among live nodes. Copyset [110] is a general-purpose replication technique that reduces the frequency of data loss. [111] designs a novel distributed layered cache system that is built on the top of the HDFS. Studies [112, 113, 114, 115] focus on solving the reliability problem for big data platforms through enhancing scheduling schemes. Replication creation strategy based on the file heat is proposed in [116] to solve the problem of uneven distribution of data in auto-sharing and hybrid clouds. Study [117] focuses on the dynamic replica placement and selection strategies in the data grid environment. [118] proposes a replication strategy based on the access pattern of files in order to optimize load balancing for large-scale user access in cloud-based WebGISs. [119] highlights the challenges involved in making a replica selection scheme explicitly cope with performance fluctuations in the system and environment. Apart from replica creation, control and management of generated replica are important to maintain the good performance of distributed storage [120]. Authors of [121] develops an automated method for identifying and repairing logical data discrepancies between database replicas in a database cluster. Triple-H [122] is a hybrid design to minimize the I/O bottlenecks in HDFS and ensure efficient utilization of heterogeneous storage devices on HPC clusters. [123] analyzes the interaction of clustering and replication in hierarchical architectures working at CDN- and RAN- level. Recently, [124] presents a formal definition of variants of the replica placement problem.

### 2.3.2 Total Cost of Ownership

Few prior studies have focused on the long-term costs of SSD-intensive storage systems so far, especially in the context of datacenters. In fact, majority of the existing literatures investigate SSD-HDD tiering storage systems. Some of them do not consider the SSD replacement cost in their total cost calculation [125, 126]. Study [127] develops a physically-accurate model of flash memory reliability to increase the endurance limit of flash. Study [128] builds a cost model with the consideration of lifetime cost of ownership including energy and power costs, and replacement cost. They assume that the “trade-in” value of the disk is a linear function of its available write cycles. Study [129] designs a tiered object store for the cloud. Their hybrid store exposes the tiering to tenants with a dynamic pricing model that is based on the tenant’s usage and the provider’s desire to maximize profits. Meanwhile, in terms of budget-driven workload allocation, [130] recently presents a systematic way to determine the optimal cache configuration given a fixed budget based

on frequencies of read and write requests to individual data items. Cui et al. [131] proposes a TCO model of datacenter considering five major costs: infrastructure, server acquisition, power utilization, networking equipment, and maintenance cost. Study [132] discusses modeling NVMe workload for optimizing TCO. However, it only addresses the online and per-I/O-request scheduling problem by minimizing the "cost" in terms of workload latencies.

# Chapter 3

# Hybrid Datacenter Storage Systems

In this chapter, we discuss the SSD-HDD hybrid datacenter storage system management problems. There are two major demands when building the storage infrastructure for a modern datacenter cluster: (1) high I/O and network throughput requirement during runtime, and (2) low latency demand for disaster recovery.

To effectively leverage flash resources to improve I/O performance, we first present an on-demand resource allocation manager (GREM) to dynamically partition and share SSD resource across multiple clients based on workload changes (Sec. 3.1). GREM divides the SSD resources into two zones to cache long-term reusable data and short-term I/O bursts, respectively. GREM also dynamically adjusts the size of each zone and each VM's reserved size in the long-term zone to effectively optimize SSD-HDD I/O updating costs.

To ensure proper replication and reliability of data from different types of disasters, we further propose a replication manager (AutoReplica) to address the reliability and accessibility issues by automatically selecting a neighbor node's SSD to backup local SSD cached data (Sec. 3.2). AutoReplica uses three approaches (i.e., ring, network, and multiple-SLA network) to automatically build/rebuild the cross-node replica structure. AutoReplica also automatically balances loads among nodes, and can conduct seamless online migration operations (i.e., migrate-on-write scheme), instead of pausing the subsystem and copying the entire dataset from one node to the other. Lastly, AutoReplica supports parallel prefetching from both primary node and replica node(s) with a new dynamic optimizing streaming technique to improve I/O performance.

## 3.1 On-Demand Flash Resource Dynamic Allocation

### 3.1.1 Motivation

In a virtualized storage system, SSDs are commonly shared by multiple VMs with heterogeneous I/O workloads and caching requirements. An effective resource management scheme should absorb hot data in SSDs, ensure good performance isolation across VMs, and maintain high resource utilization of SSDs. To achieve this goal, we need to thoroughly understand I/O access patterns of heterogeneous VM workloads and dynamically allocate SSDs among these VMs according to I/O workload changes. Therefore, in this subsection, we present our analysis of I/O access patterns in real production systems, and discuss the limitations of two straightforward approaches.

#### 3.1.1.1 Understanding I/O Access Patterns

We studied a suite of real I/O traces to analyze and understand I/O access patterns in enterprise production systems.

**[MSR Cambridge]** One week I/O block traces collected by MSR Cambridge in 2007 [133]. In these I/O traces, each data entry describes an I/O request, including timestamp, disk number, logical block number (LBN), number of blocks and the type of I/O (i.e., read or write). There are 36 traces from MSR-Cambridge, which includes a variety of workloads.

**[FIU]** Two sets of I/O block traces collected by Florida International University (FIU) [83]. *FIU IODedup* contains collected downstream of an active page cache for three weeks in 2008. *FIU SRCMap* covers I/O accesses from an email server, a virtual machine monitor running two web servers, and a file server workload during 2008-2009.

**[UMASS]** Two financial I/O traces (*Fin1* and *Fin2*) from OLTP applications running at large financial institutions and three I/O traces (*WebSch1*, *WebSch2* and *WebSch3*) from a web search engine [134].

Table 3.1 shows some statistical results of selected I/O traces, including:

- *Hit Ratio:* the percentage of I/Os that are hit in SSDs under the LRU algorithm with a fully associative cache of $4KB$ cache line and $1GB$ cache size.
- *Working Volume ($WV$) Size:* the total amount of data (in bytes) accessed in the disk.
- *Working Set ($WS$) Size:* the total address range (in bytes) of accessed data, which is the unique set of $WV$. A large working set covers more disk space. If the cache size is larger than or equal to

| Group | Trace Name | Hit (%) | WV (GB) | WS (GB) | Seq (%) | Wr (%) | Peak IOPS |
|---|---|---|---|---|---|---|---|
| MSR-F1 | mds0 | 90.84 | 67.83 | 6.43 | 32.50 | 88.11 | 207.02 |
| | stg0 | 89.28 | 21.63 | 13.21 | 42.43 | 84.81 | 187.01 |
| | usr0 | 88.25 | 31.81 | 7.49 | 64.38 | 59.58 | 138.28 |
| | src12 | 85.63 | 16.00 | 5.14 | 42.45 | 74.63 | 143.51 |
| MSR-F2 | hm0 | 91.34 | 27.88 | 9.03 | 33.76 | 64.50 | 271.65 |
| | prn0 | 85.16 | 132.65 | 32.74 | 38.71 | 89.21 | 254.55 |
| | web0 | 77.14 | 67.82 | 14.91 | 40.37 | 70.12 | 249.67 |
| | web1 | 54.25 | 135.66 | 8.68 | 84.57 | 45.89 | 146.44 |
| MSR-U | stg1 | 34.59 | 203.47 | 162.03 | 85.64 | 36.25 | 197.75 |
| | usr2 | 19.48 | 1060.78 | 763.12 | 77.64 | 18.87 | 584.50 |
| | web2 | 6.2 | 339.16 | 152.65 | 85.43 | 85.43 | 0.75 |
| | src21 | 2.82 | 339.15 | 41.63 | 89.50 | 2.14 | 303.64 |
| FIU-F1 | wbusr1 | 83.78 | 15.22 | 0.32 | 62.29 | 99.97 | 1.83 |
| | wbrsh5 | 74.92 | 15.75 | 0.41 | 73.23 | 100.00 | 31.64 |
| | wbmal4 | 70.8 | 36.48 | 5.87 | 48.28 | 85.20 | 136.25 |
| | wbmal5 | 70.18 | 36.35 | 4.39 | 49.89 | 87.26 | 156.63 |
| FIU-F2 | hm4t3 | 83.38 | 283.40 | 1.65 | 87.22 | 99.91 | 467.58 |
| | hm4t1 | 76.97 | 264.81 | 1.67 | 41.95 | 92.66 | 118.98 |
| | wbusr3 | 75.98 | 15.75 | 3.55 | 69.11 | 87.48 | 129.10 |
| | mal1c1 | 72.04 | 557.74 | 36.27 | 94.96 | 88.35 | 488.93 |
| FIU-U | hm2i1 | 41.41 | 258.32 | 15.68 | 61.77 | 76.31 | 284.06 |
| | hm2i2 | 28.04 | 391.50 | 5.27 | 74.19 | 91.45 | 371.74 |
| | hm3m3 | 26.79 | 37.25 | 0.48 | 82.65 | 99.17 | 39.36 |
| | hm3m2 | 17.23 | 36.49 | 0.04 | 80.27 | 99.85 | 1.88 |
| UMass-F | Fin1 | 99.07 | 1289.06 | 1.08 | 32.52 | 76.84 | 218.59 |
| | Fin2 | 98.51 | 1.16 | 1.11 | 11.79 | 17.65 | 159.94 |
| UMass-U | WebSch1 | 6.08 | 33.35 | 18.37 | 1.78 | 0.02 | 355.38 |
| | WebSch2 | 6.3 | 33.35 | 18.98 | 3.71 | 0.02 | 375.02 |
| | WebSch3 | 6.15 | 33.35 | 19.21 | 14.62 | 0.03 | 245.09 |

Table 3.1: Statistics for selected *MSR-Cambridge*, *FIU* and *UMASS* workloads.

a workload's $WS$, then the I/O hit ratio of this workload can be close or equal to 100% under the LRU caching algorithm [135].

- *Sequential Ratio ($Seq$):* the amount (in bytes) of total sequential I/Os (both read and write) divided by the total I/O amount (in bytes). In general, SSDs have better performance under sequential I/Os than under random I/Os.
- *Write I/O Ratio ($Wr$):* the number of write I/Os divided by the total number of I/Os.
- *Peak Throughput (IOPS):* the peak throughput of the I/O workload (with the sampling window of $5min$).

As shown in Table 3.1, high variance can be found in I/O hit ratios across different I/O workloads. For example, the I/O hit ratio is more than $90\%$ under the `MSR-hm0` workload while the I/O hit ratio under the `MSR-src21` workload is less than $3\%$. We thus coarsely classify these I/O workloads into two categories: (1) *cache-friendly* workloads (e.g., `MSR-hm0`, `UMASS-Fin1`) always obtain high I/O hit ratios, while (2) *cache-unfriendly* workloads (e.g., `MSR-web2`, `FIU-hm2i1`) have relatively low I/O hit ratios. We observe that the working set (i.e., the unique data blocks) in cache-friendly workloads (denoted as "F" in Table 3.1) is usually small, which indicates high spatial locality (i.e., high reuse ratio, defined as $\frac{WV}{WS}$), and thus is highly likely to be cached and hit in SSDs. In contrast, cache-unfriendly workloads (denoted as "U" in Table 3.1) often have large working volume sizes (see the $WV$ column in Table 3.1) and working set sizes (see the $WS$ column in Table 3.1). This observation motivates that we should differentiate these two classes of workloads by reserving a particular amount of SSD resources for VMs that have cache-friendly workloads to hold their popular data blocks. Moreover, the reserved SSDs do not need to be too large to guarantee high hit ratios of VMs with cache-friendly workloads since their working set sizes are usually small.

Fig. 3.1 further shows the runtime working set sizes ($WS$) of two MSR workloads during every 5min epoch. We observe that cache-unfriendly workloads (see plot (b) in Fig. 3.1) have more I/O spikes (i.e., a large amount of unique data blocks accessed during a short period of time) than cache-friendly workloads (see plot (a) in Fig. 3.1). These I/O spikes in cache-unfriendly workloads are much more frequent, which can dramatically degrade I/O hit ratios due to the first-time cache miss and even worse pollute the critical data in SSDs. This observation implies that VMs with cache-unfriendly workloads need to be assigned with a large amount of SSDs during their bursty periods to absorb and handle their bursty I/Os for improving their hit ratios. However, to avoid severe cache pollution, allocated SSDs should not overlap the reserved SSDs for cache-friendly workloads.

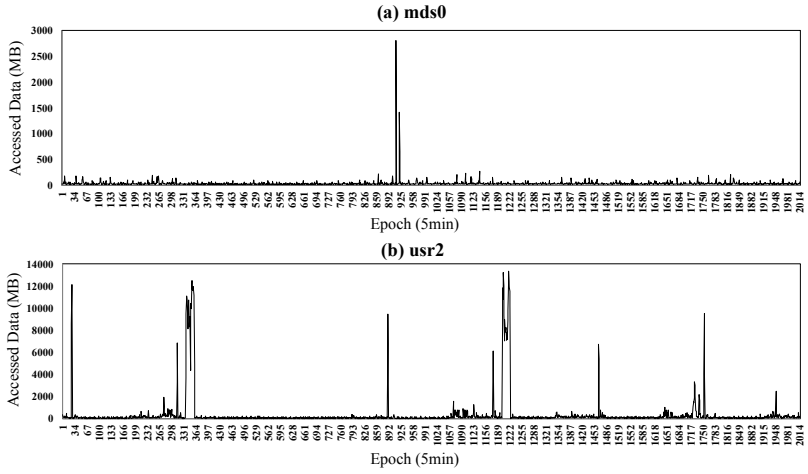



Figure 3.1: Examples of bursty I/Os (e.g., runtime working set sizes) under (a) cache-friendly workload `mds0` and (b) cache-unfriendly workload `usr2`.

Therefore, a new SSD resource management scheme is needed to discriminate cache-friendly and cache-unfriendly workloads and improve I/O performance for both types of workloads.

#### 3.1.1.2 Limitations of Straightforward Approaches

Two straightforward approaches can be used to allocate SSD resources among multiple VMs. The first approach (referred to as "*performance isolation*") is to proportionally reserve SSD resources for each VM in the system such that all VMs are purely isolated in using their own assigned SSD resources. Different cache replacement algorithms (e.g., LRU [52], CAR [54], DellFluid [136], Mercuy [137], and SCE [138]) can be used by each VM to cache their recently accessed data blocks and the caching management is fully affected by their own workload changes. In contrast, the second approach [139, 140] (referred to as "*fair competition*") manages SSD resources in a fair competition mode by allowing all VMs to freely use or share the entire SSDs. A caching algorithm is usually used to centrally decide which data blocks should be held in SSDs for all VMs. Consequently, the caching management is inevitably interfered by the intensity of all workload changes.

However, we find that neither of these approaches can fully utilize the benefits of SSDs when some of VMs have bursty I/O workloads during runtime. To understand the limitations of these two approaches, we run trace-driven experiments by replaying a mix of real I/O workloads and investigating the assignment of SSD resources to each VM (or each workload) across time duration about one week. We find that although the first approach (i.e., *performance isolation*) is able to avoid

performance interference, VMs with bursty I/Os, unfortunately, have no chance to obtain more SSD resources during their bursty periods. Each VM keeps the fixed amount of SSDs during their runtime.

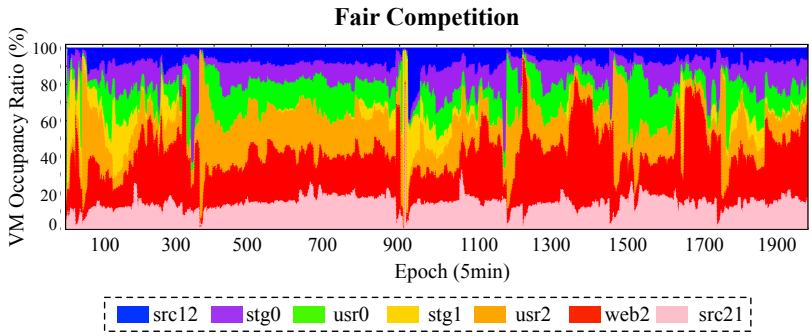



Figure 3.2: Runtime SSD allocation among VMs under *fair competition*.

On the other hand, the second approach solves this issue by allowing all VMs to compete SSDs based on their present I/O demands. As shown in Fig. 3.2, VMs (e.g., cache-unfriendly workloads `web2` and `usr2`) occupy more SSD resources when there are I/O spikes in their workload. Thus, their I/O hit ratios are improved and the overall utilization of SSD resources is increased as well. However, we notice that under this approach, VMs with bursty I/Os might occupy a large amount of the SSD resources (e.g., `web2` at epoch 900 in Fig. 3.2) during their bursty periods by evicting other cached data, which might pollute critical caching of VMs with cache-friendly workloads and then degrade their I/O hit ratios.

### 3.1.2 Dynamic Partition Manager

#### 3.1.2.1 Basic Idea of GREM

As observed in Sec. 3.1.1, VMs with cache-friendly workloads can usually achieve high hit ratios when only a small amount of their critical data blocks (i.e., their working sets) are cached in SSDs. However, VMs with cache-unfriendly workloads often have spikes (i.e., a large amount of unique data blocks) of I/Os across time, which can incur a significant amount of cache misses and further pollute the critical caching of other VMs. In order to ensure all VMs benefit from SSDs, we design a new resource management scheme, named GREM, which strives to discriminate different workload types (e.g., cache-friendly and cache-unfriendly workloads) by splitting SSDs into two zones (denoted as "$Z_L$" and "$Z_S$" see Fig. 3.3) such that one zone is designed for reserving SSD resources for each VM and the other zone is used to absorb and handle bursty I/Os. GREM manages

SSD resources underlying the VM page buffer layer, thus it does not need to (reversely) interact with the VM page buffers.

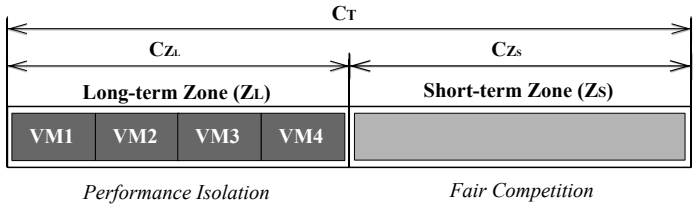



Figure 3.3: Basic structure of GREM.

In particular, we refer "$Z_L$" to as a "*Long-term Zone*", which is expected to cache the most popular data blocks for each VM based on their I/O access frequency during a long period. Furthermore, SSD resources in "$Z_L$" are reserved for each VM such that their critical and popular data blocks can be kept in SSDs without any pollution from other VMs, which thus guarantees a high hit ratio from VMs with cache-friendly workloads. We refer "$Z_S$" to as a "*Short-term Zone*", where SSD resources are fairly competed among VMs based on the popularities of their recently accessed data during a short period. Consequently, VMs with cache-unfriendly workloads can have a high chance to get SSD resources in $Z_S$ to cache data for their bursty I/Os and achieve an improved I/O hit ratio. Given the total capacity of SSD resources $C_T$, we have

$$C_T = C_{Z_L} + C_{Z_S}, \tag{3.1}$$

where $C_{Z_L}$ and $C_{Z_S}$ denote the capacities of $Z_L$ and $Z_S$, respectively. Furthermore, given $m$ VMs running in the system, each VM $i$ will be assigned with $C_{Z_L}(i)$ SSD resources in $Z_L$ such that $\sum_{i=1}^{m} C_{Z_L}(i) = C_{Z_L}$. Therefore, the design goal of GREM is to determine the proper values of $C_{Z_L}$, $C_{Z_S}$, and $C_{Z_L}(i)$ in order to maximize the overall I/O hit ratio and minimize the I/O cost for operating I/O access and updating SSD content.

#### 3.1.2.2 Dynamically Partition of the Long-term Zone

As introduced in Sec. 3.1.2.1, GREM attempts to reserve SSD resources in $Z_L$ for each VM in order to ensure each VM have their own private SSDs to cache their critical hot data. One straightforward approach is to partition zone $Z_L$ among VMs equally or proportionally, i.e., $C_{Z_L}(i) = C_{Z_L} \cdot w_i$, where $w_i$ is a fixed weight based on the SLA for VM $i$ and $\sum_{i=1}^{m} w_i = 1$, and reserve a fixed amount (i.e., $C_{Z_L}(i)$) of SSDs in $Z_L$ for VM $i$. However, we found that this approach

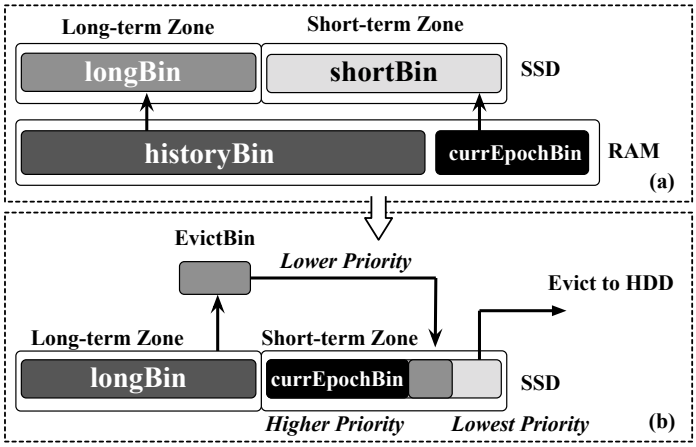



Figure 3.4: High level idea of GREM: (a) cache admission, and (b) cache eviction.

is ineffective when workloads frequently change and spikes of I/Os occur across time. Reserved SSD resources cannot be fully utilized when some VMs start to have a low I/O access rate and meanwhile other VMs may not be able to obtain sufficient SSDs when they experience bursty I/Os that need to access a large amount of unique data blocks (i.e., large working sets). Therefore, we develop a dynamic partitioning algorithm for GREM to dynamically decide the capacity (i.e., $C_{Z_L}(i)$) for each VM's reserved SSD space in $Z_L$ based on not only each VM's access history in a long term but also their I/O workload changes. Here, we assume that the capacities of two zones are fixed, e.g., $C_{Z_L} = C_{Z_S} = \frac{C_T}{2}$. Later, we present a new version of GREM that adjusts the zone sizes in an online mode.

**[Dynamic $Z_L$ Partition and Cache Replacement Solution]** Alg. 1 describes the procedure of GREM, including how GREM periodically updates the content in both $Z_L$ and $Z_S$, and how GREM online adjusts the amount of reserved SSDs in $Z_L$ for each VM. $historyBin$ is a dictionary in which key is bin IDs of all VMs and value is the relative access count for each bin. $currEpochBin$ is the accessed bins of all VMs in the current epoch. $shortBin$ and $longBin$ are the cached bins of all VMs in $Z_S$ and in $Z_L$, respectively. $flashBin$ is those bins need to be cached in the SSD. Fig. 3.4 further shows the procedures of GREM for cache admission and cache eviction. The key idea of GREM is that the amount of reserved SSDs in $Z_L$ for each VM should be proportional to their long-term access behaviors. When the distribution of data bin popularities changes, GREM dynamically adjusts the reservation of SSD resources for each VM in order to fully utilize the $Z_L$ zone. In detail, GREM maintains a long-term I/O access history for all running VMs (i.e., "$historyBin$" in Fig. 3.4) to record the accumulative I/O popularity statistics for their bins (e.g., each bin size is $1MB$). This

**Algorithm 1:** Dynamic Partition in $Z_L$

```
1  Procedure GREM()
2      UpdateLongTermZone();
3      UpdateShortTermZone();
4      flashBin = shortBin + longBin;
5      return flashBin;
6  Procedure UpdateLongTermZone()
7      if size(historyBin) ≤ size(longBin) then
8          longBin = historyBin.keys;
9      else
10         /* the max number of bins to be cached in Z_L */ j = size(longBin);
11         /* the top j popular bins */
12         itemH = number of j bins in historyBin.keys with highest historyBin.values;
13         /* the evicted bins due to newly cached bins */ evictBin = bins of longBin which
             are not in itemH;
14         longBin = itemH;
15     return;
16 Procedure UpdateShortTermZone()
17     if size(longBin) < size(historyBin) ≤ C_T then
18         /* Case 1: warming up period */
19         shortBin = the remaining bins of historyBin.keys which are not in longBin;
20     else if size(historyBin) > size(flashBin) then
21         /* Case 2: historyBin, evictBin, currEpochBin and existing bins in shortBin
             compete in the Z_S */
22         shortBin −= bins of shortBin which are also in longBin;
23         currEpochBin −= bins of currEpochBin which are also in longBin;
24         if size(currEpochBin) ≥ size(shortBin) then
25             j = size(shortBin);
26             shortBin = number of j bins in currEpochBin with highest I/O popularity;
27         else
28             shortBin += evictBin;
29             shortBin −= bins of shortBin which are also in currEpochBin;
30             j = size(shortBin) − size(currEpochBin);
31             shortBin = number of j bins in shortBin with highest I/O popularity;
32             shortBin += currEpochBin;
33     return;
```

historical information can be saved in the RAM. Similarly, these long-term I/O access records are incrementally updated every time epoch (e.g., every 5 min). An aging function is also used to capture the variation of I/O popularities over time. Recent I/O popularities (i.e., collecting I/O activities in recent time epochs) are assigned with higher weights for contributing more to the accumulative I/O popularity statistics. Once the I/O popularity statistics are updated, GREM selects the most popular bins with their overall size is equal to $C_{Z_L}$, and then sets the amount of reserved SSDs for VM $i$, i.e., $C_{Z_L}(i)$, to the total size of its popular bins that have been selected and cached in $Z_L$ (see Fig. 3.4(a) and lines 6-15 in Alg. 1). By this way, GREM also updates the content of SSDs in $Z_L$ by caching the most popular bins with total size of $C_{Z_L}$. On the other hand, GREM attempts to leverage the recent data access information (i.e., during the current epoch) to update the short-term zone $Z_S$ (see Fig. 3.4(b) and lines 16-33 in Alg. 1). In particular, GREM also records the I/O popularity statistics for all bins that were accessed in the last epoch (i.e., "$lastEpochBin$" in Fig. 3.4) in the RAM. The most popular bins that have not been cached in $Z_L$ are then cached in $Z_S$. The total size of these cached bins is bounded by $C_{Z_S}$. We notice that it is possible that all accessed bins in the "$lastEpochBin$" might have the total size less than $C_{Z_S}$. In such a case, GREM further considers to cache the bins that are just evicted from $Z_L$.

#### 3.1.2.3 Dynamical Partition for $Z_L$ and $Z_S$: D_GREM

GREM statically and equally partitions SSDs into two zones, i.e., $C_{Z_L} = C_{Z_S} = \frac{C_T}{2}$. However, we find that such a partitioning unfortunately may not be optimal to general cases. For example, if workloads have a large number of bins being popular only during a short period, then $Z_S$ may not be large enough to handle bursty I/Os that access those bins. This can cause a very low I/O hit ratio and increase the operational costs for caching new bins in $Z_S$. To solve this problem, we design a bursty-detection based partition algorithm that allows GREM to dynamically adjust the sizes of $Z_L$ and $Z_S$. We refer this new version of GREM to as D_GREM. Fig. 3.5(a) depicts the high level sketch of D_GREM consisting of two main components, i.e., *bursty detector* and *strategy switcher*. The *bursty detector* takes the feedbacks of workload changes and cache performance (e.g., I/O hit or I/O miss) as the input to determine if the current workload is bursty or non-bursty. Based on the detected result, the *strategy switcher* will make different partitioning decisions for improving the SSD resource utilization.

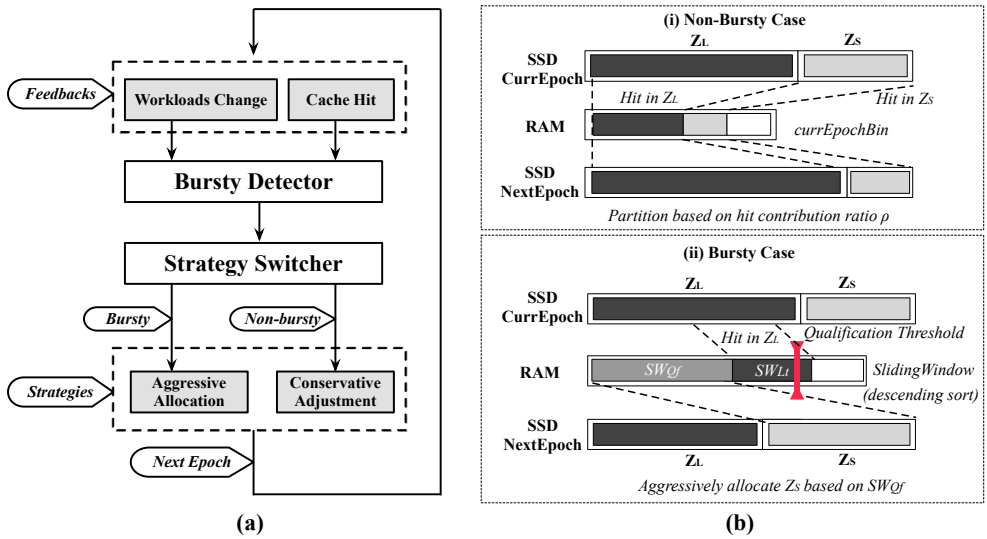



Figure 3.5: (a) The high level sketch of D_GREM. (b) Dynamic partition of $Z_L$ and $Z_S$ of D_GREM.

**3.1.2.3.1 Bursty Detection** Basically, bursts or spikes contain a relatively high number of I/O accesses on a large amount of working set data. They often occur within a short time period. The bursty detector is expected to help D_GREM decide (1) when to adjust the partition of SSDs, and (2) how to divide SSD resources between two zones. We find that the capacity adjustment of two zones is needed when the number of popular bins or the working set size significantly changes. Intuitively, the new capacity of each zone should be related with the number of popular bins in the current spike, the bin reuse rates, the bin access history during previous epochs, and the hit counts in the $Z_L$ and $Z_S$. Therefore, D_GREM uses a sliding window ($SW$), e.g., 5 min, to record I/O access history for all VMs in recent epochs. Again, this historical information can be kept in the RAM. D_GREM tracks the changes in the working set sizes between the current (i.e., $|WS_{curSW}|$) and previous (i.e., $|WS_{prevSW}|$) sliding windows. The relative difference between these two sliding windows is then defined as bursty degree (denoted as $B_d$) as follows.

$$\begin{aligned} B_d &= \Delta(WS_{curSW}, WS_{prevSW}) \\ &= \frac{||WS_{curSW}| - |WS_{prevSW}||}{|WS_{curSW}|}. \end{aligned} \tag{3.2}$$

The value of $B_d$ is a good indicator of working set changes across time. The bursty detector claims the arrival of bursty I/Os when the value of $B_d$ is beyond a predefined threshold $\beta$. We set $\beta$ to 0.6 in our experiments.

**3.1.2.3.2 Non-Bursty Case Strategy** When there is no burstiness in current I/O workloads, i.e., $B_d < \beta$, D_GREM tunes the splitting point between two zones (i.e., $Z_L$ and $Z_S$) by leveraging the feedback of each zone's caching performance under the present partition (as shown in Fig. 3.5(b)(i)). In particular, we evaluate the importance of two zones (i.e., their contributions to the overall I/O performance) by recording the total I/O hit volumes (i.e., the amount of all cached data that are hit by one or multiple I/Os) in each zone during the recent epoch (e.g., 5 min). We define a contribution ratio $\rho$ as follows:

$$\rho = \alpha \times \frac{HV_L}{HV_S}, \tag{3.3}$$

where $\alpha \in [0, 1]$ is a tunable parameter of importance, and $HV_L$ and $HV_S$ represent the I/O hit volumes of $Z_L$ and $Z_S$, respectively. By default, $\alpha = 0.35$. Once $\rho$ is updated at the end of an epoch, D_GREM adjusts the capacities of two zones for next epoch using Eq.(3.4).

$$\begin{cases} C_{Z_S} = \frac{C_T}{1+\rho}, \\ C_{Z_L} = C_T - C_{Z_S}, \end{cases} \tag{3.4}$$

where $C_T$ is the capacity of SSDs and $C_{Z_S}$ and $C_{Z_L}$ are the new anticipated capacities of $Z_L$ and $Z_S$, respectively. Intuitively, the zone that contributes more to the overall hit ratio is highly likely to get more SSD resources and the allocation of SSDs is proportional to the contribution ratio.

**3.1.2.3.3 Bursty Case Strategy** When bursty I/Os are identified by the bursty detector, D_GREM turns to aggressively shift SSD resources from one zone to the other. Using the contribution ratio as a feedback to reset the capacities of two zones unfortunately does not work well under this case because this approach cannot quickly adapt to workload changes. The corresponding delay can further incur "cascade effect" (also called "thrashing effect") of insufficient capacity in one of the zones. For example, when bursty I/Os that access new bins arrive, the I/O hit volume of $Z_S$ in the current epoch might not be large enough to get more SSDs for handling those bursty I/Os. Consequently, this zone's I/O hit volume becomes even less in the next epoch, which indicates less importance and then keeps reducing the capacity of $Z_S$ if we use Eq.(3.4).

To avoid such a cascade effect, we attempt to dynamically and aggressively assign more SSDs to $Z_S$ when bursty I/Os are found in workloads, and further to minimize the penalty on $Z_L$'s caching performance. The general idea of our approach is that if the working set size of accessed bins in the current epoch increases dramatically compared with the previous epoch, then it would also be helpful if we increase the size of $Z_S$ for absorbing the spikes in the near future. Meanwhile,

we should ensure those popular bins, which are cached in $Z_L$ and are recently hit, to keep staying in $Z_L$, as shown in Fig. 3.5(b)(ii).

**Algorithm 2:** Dynamic Partition of $Z_L$, $Z_S$: D_GREM

```
Output: Partitioning solution ⟨C_{Z_L}, C_{Z_S}⟩
1  Procedure ResizeFlashBin()
2      B_d = ||WS_{curSW}| − |WS_{prevSW}|| / |WS_{curSW}|
3      if B_d > β then
4          C_{Z_S} = GetHitContributionRatio();
5      else
6          ρ = GetHitContributionRatio();
7          C_{Z_S} = 1/(1 + ρ) × C_T;
8      Round Up to Boundaries C_{Z_L} ∈ [B_L, B_U];
9      Round Up to Boundaries C_{Z_L} = C_T − C_{Z_S} ∈ [B_L, B_U];
10     return ⟨C_{Z_L}, C_{Z_S}⟩;
11 Procedure GetQualifiedShortTermBinSetSize()
12     B_{SWLt} = SW ∩ Z_L;
13     Th_{SW} = γ × B̄_{SWLt};
14     B_{SWQf} = {x | x ∈ SW, x ∉ B_{SWLt}, x > Th_{SW}};
15     return size(B_{SWQf});
16 Procedure GetHitContributionRatio()
17     Get HV_L and HV_S;
18     return HV_L / HV_S;
```

Therefore, we use the sliding window ($SW$) to record I/O popularity statistics for all bins that are accessed in recent several epochs (instead of the latest one). D_GREM identifies all bins that are recorded in the current sliding window and are also cached in the $Z_L$ zone. We refer this set of bins to as "$B_{SWLt}$". D_GREM uses the average number of accesses ($\overline{B_{SWLt}}$) of all bins in $B_{SWLt}$ as a criterion to set the threshold ($Th_{SW}$) for choosing hot bins to be cached in $Z_S$ as follows.

$$Th_{SW} = \gamma \times \overline{B_{SWLt}}, \tag{3.5}$$

where $\gamma$ is an adjustment parameter, and is set to 1.2 by default. D_GREM then finds all the bins that have been accessed in the current sliding window more than $Th_{SW}$ times but are not currently cached in $Z_L$. We refer this set of "qualified" bins to as "$B_{SWQf}$", as:

$$B_{SWQf} = \{x | x \in SW, x \notin B_{SWLt}, x > Th_{SW}\}. \tag{3.6}$$

We then set the anticipated capacity of $Z_S$ to the total size of bins in $B_{SWQf}$.

$$C_{Z_S} = |B_{SWQf}| \,. \tag{3.7}$$

**3.1.2.3.4 Dynamic Tuning Procedure** As shown in Fig. 3.5(a), initially, capacities of both two zones are set to half of the entire SSDs. D_GREM recalculates $B_d$, the present bursty degree at the end of each sliding window and determines if bursty I/Os are arriving by comparing with the threshold $\beta$. Under the bursty case, D_GREM aggressively enlarges the capacity of $Z_S$ according to Eq.(3.7). On the other hand, D_GREM bases on the contribution ratio of two zones to adjust the allocation of SSDs to these zones when there is no burstiness, as shown in lines. A boundary checking is further considered to ensure the minimum capacity for each zone.

Alg. 2 summarizes the main idea of how D_GREM dynamically adjusts the partition between $Z_L$ and $Z_S$ for improving overall I/O hit ratio. Initially, capacities of both two zones are set to half of the entire SSD resource. D_GREM recalculates $B_d$, the present bursty degree at the end of each sliding window and determines if bursty I/Os are arriving by comparing with the threshold $\beta$, see line 2. Under the bursty case, D_GREM aggressively enlarges the capacity of $Z_S$ according to Eq.(3.7), see lines 11-14. On the other hand, D_GREM bases on the contribution ratio of two zones to adjust the allocation of SSDs to these zones when there is no burstiness, as shown in lines 6-7. A boundary checking is further considered to ensure the minimum capacity for each zone, see line 8 in the algorithm.

### 3.1.3 Evaluation

In this subsection, we conduct a trace-driven evaluation by replaying real enterprise I/O workloads. We implement two versions of our proposed algorithm: (1) GREM that assigns $50\%$ of the total SSDs to each of two zones, and adaptively adjusts partitions of each VM in $Z_L$ according to the workload change; and (2) D_GREM that further dynamically adjusts the sizes of $Z_L$ and $Z_S$ during runtime. For comparison, we also implement conventional caching algorithms, such as global LRU (GLRU) [52], global CAR [54], and a recently proposed tiering algorithm VFRM [139]. We evaluate the effectiveness of GREM with respect to our primary goals, i.e., maximizing the I/O hit ratio and minimizing the I/O cost incurred in managing SSDs.

#### 3.1.3.1 Testbed and Implementation Details

Our evaluation environment is calibrated based on the real testbed specs summarized in Table 3.2. In detail, to deal with the management across physical nodes and multiple tiers, we adopt the same hybrid file developed in our previous work [139], which consists of two "files": a *base file* on the spinning disk tier and a corresponding *peer file* on the flash tier.

| Component | Specs |
|---|---|
| Processor | Intel Xeon E5-2430 2.20GHz |
| Processor Cores | 6 Cores |
| Memory Capacity | 48GB ECC DDR3 R-DIMMs |
| Memory Bandwidth | $102.4GB/s$ |
| RAID Controller | LSI SAS 2008 |
| Network | 10 Gigabit Ethernet NIC |
| Operating system | Ubuntu 12.04.5 |
| Linux Kernel | 3.14 Mainline |
| SSD Model | Intel and Samsung NVMe SSDs (2015) |
| HDD Model | Western Digital 20EURS-63S48Y0 (5400 RPM) |
| HDD Capacity | 2 TB |

Table 3.2: Testbed setup and specification.

Furthermore, we use the following three key techniques to lower the overhead: (1) a "*Pointer Block Cache*" of the peer file is used as the block look-up table, which can eliminate the need for an extra lookup table; (2) a "*Heat Map*" is used to represent the I/O popularity statistics of each "1MB block" of the files on VMFS, and 1MB block only requires 16 bytes to hold the popularity stats, which is only 0.0015% of the size of the VMDK; and (3) a "*Tiering Map*" is used to represent the placement of the blocks between tiers, whose space overhead is about 0.00001% of the size of the VMDK. Moreover, both the heat and tiering maps do not need to be pinned in memory permanently. Lastly, based our observation in Sec. 3.1.1 and our offline sensitivity analysis results, we set 1MB and 5min as spatial and temporal granularities, respectively, to avoid adverse impact from spikes as well as to reduce the caching management overhead.

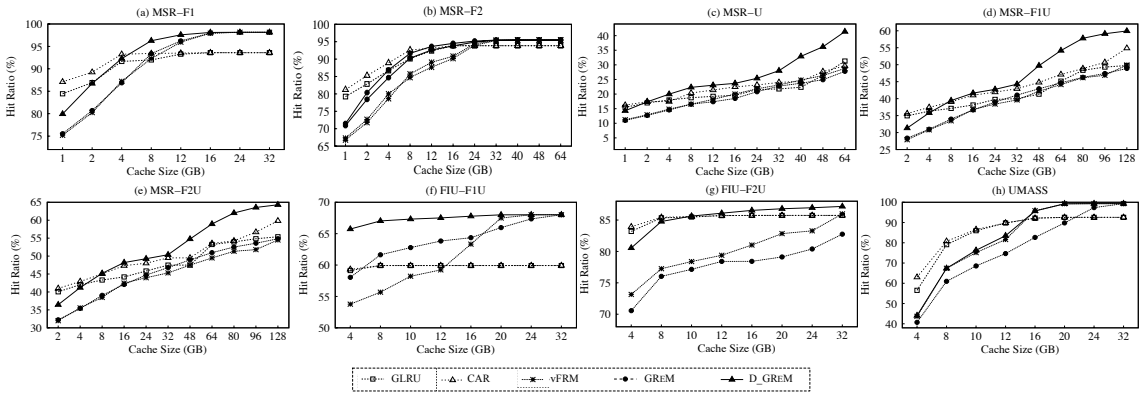



Figure 3.6: I/O hit ratio results of workload combinations of MSR, FIU and UMASS repositories under different cache sizes and cache algorithms.

#### 3.1.3.2 I/O Hit Ratio

Fig. 3.6 shows the overall (i.e., read & write) I/O hit ratios as a function of SSD cache size (in GB) under diverse workloads that are mixed with different MSR, FIU and UMASS repositories. For example, "FIU−F1U" is a mixed workload with the I/O traces of "FIU−F1" and "FIU−U", where "F" and "U" refer to cache-friendly and cache-unfriendly workloads, as summarized in Table 3.1. In overall, we can see that GREM and D_GREM are superior to other existing caching algorithms. For instance, Fig. 3.6(a) presents the results under MSR−F1, a cache friendly workload. When the cache size is smaller than $2GB$, VFRM, GREM and D_GREM all have lower I/O hit ratios than the conventional algorithms. However, as the capacity of SSDs increases to $4GB$, D_GREM outperforms GLRU and CAR. When we have more than $8GB$ SSDs, the I/O hit ratios under both GREM and D_GREM are beyond those of GLRU and CAR. More importantly, the conventional algorithms cannot take advantage of a large SSD cache. Their I/O hit ratios reach the converging point (i.e., about 94%) when the cache size is 4GB. In contrast, GREM and D_GREM can further use the benefits of the increasing SSD capacity to improve I/O hit ratios up to 98%. Moreover, due to large working set sizes and bursty I/Os, most caching algorithms, including GREM, cannot achieve high I/O hit ratios for cache-unfriendly workloads (e.g., MSR−U in Fig. 3.6(c)) even when we increase the SSD capacity. By dynamically adjusting the size of the short-term zone ($Z_S$) to absorb bursty I/Os, D_GREM keeps improving I/O hit ratio up to 40% when we have 64GB SSD space. We further notice that D_GREM still does not converge even with 64GB SSD space, which indicates that this algorithm is able to further achieve better I/O hit ratios.

#### 3.1.3.3 I/O Cost

Fig. 3.7 shows the normalized overall I/O costs (SSD/HDD access and SSD-HDD contents updating latencies [140]) based on the measured data from an $80GB$ `Intel DC S3500 Series` SSD and a $2TB$ 5400 RPM `Western Digital WD20EURS-63S48Y0` HDD, e.g., I/O latencies of SSD/HDD read and write operations under $4KB$ and $128KB$ cache lines.

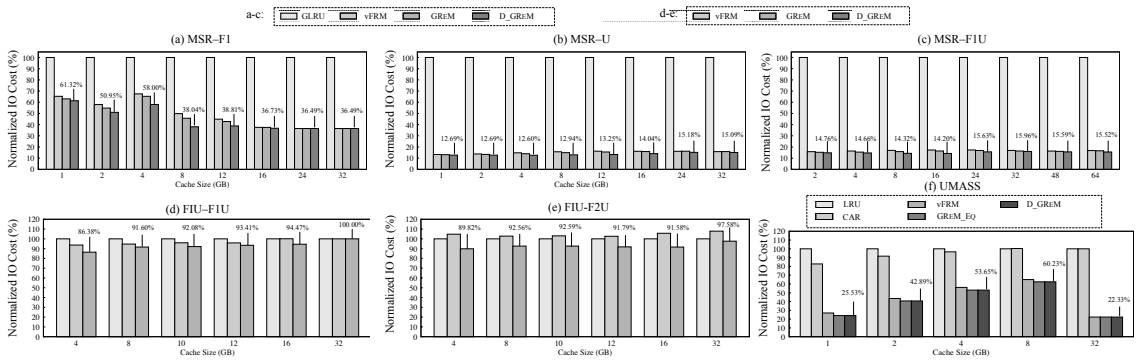



Figure 3.7: Normalized I/O cost of `MSR`, `FIU` and `UMASS` workloads, which is total latencies of read/write operations of SSD and HDD.

In our evaluation, the conventional caching algorithms (i.e., GLRU and CAR) use $4KB$ as the cache line size, while other algorithms (i.e., vFRM, GREM and D_GREM) use the cache line size of $128KB$ and group I/Os into $1MB$ bins. We observe that by using the coarse-update granularity, all these algorithms (e.g., vFRM and D_GREM) significantly reduce the overall I/O costs compared to the conventional caching solutions, especially when there are cache-unfriendly workloads (e.g., `MSR-U` and `MSR-F1U`). Furthermore, D_GREM always achieves the lowest cost in all these cases. For example, as shown in Fig. 3.7(a), with $8GB$ cache size, the overall I/O costs of `MSR` with four cache-friendly workloads under D_GREM are $61.96\%$ lower than GLRU, $11.77\%$ lower than vFRM and $7.62\%$ lower than GREM. Similar observations can be found under cache-unfriendly workloads, see Fig. 3.7(b).

In Figs. 3.7(c)-(d), we further investigate the I/O costs under different algorithms when we have workloads mixed with both cache-friendly and cache-unfriendly workloads. Again, by dynamically adjusting the sizes of $Z_L$ and $Z_S$ and updating SSD content in the coarse granularity, D_GREM significantly reduces cache pollution due to I/O spikes, and avoids too frequent SSD content updates, which thus achieves much lower I/O costs compared to the conventional caching algorithms, e.g., GLRU and CAR. Both GREM and D_GREM further slightly reduce the I/O costs

compared to vFRM, which also adopts the coarse granularity for SSD content updating, but cannot avoid bursty I/Os evicting the cached critical data.

#### 3.1.3.4 Dynamic Partition Inside $Z_L$

We track the SSD occupancy ratio of each VM for every 5 minutes under GREM to see how GREM can dynamically adjust the amount of SSDs allocated to each VM during the runtime. We show three representative results from MSR workloads in Figs. 3.8(a)-(c). In this set of experiments, the capacities of $Z_L$ and $Z_S$ are fixed to be half of the total cache size, i.e., (a) and (b) are $1.5GB$, and (c) is $3GB$. As shown in Fig. 3.8(a), cache-friendly workloads are relatively stable, thus the amounts of reserved SSDs to all four VMs do not change dramatically across the time.

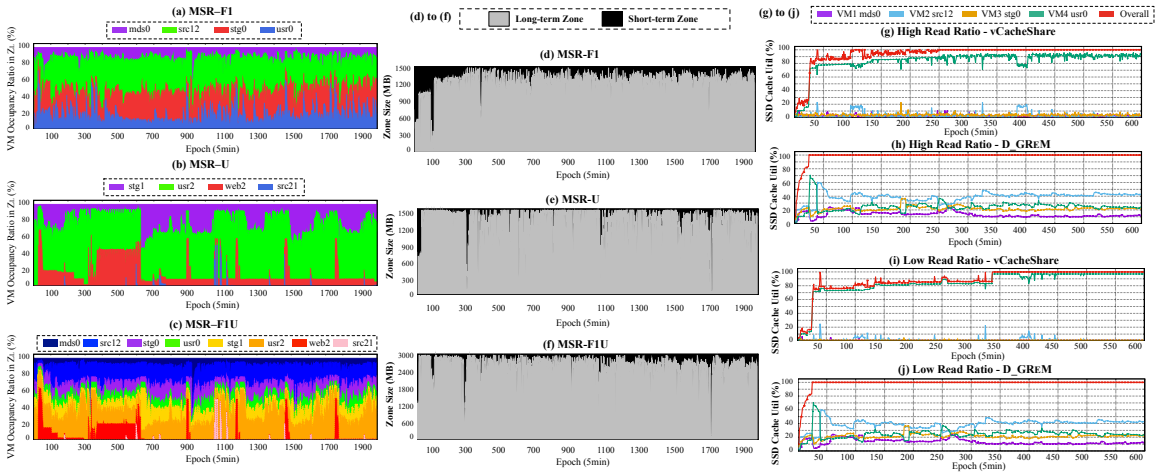



Figure 3.8: (a)-(c): Partition inside $Z_L$ for MSR−F1, MSR−U and MSR−F1U workloads under GREM. (d)-(f): Partition between $Z_L$ and $Z_S$ for MSR−F1, MSR−U and MSR−F1U under D_GREM. (g)-(j): Cache partitioning effectiveness: D_GREM vs. vCacheShare under workloads with different read-write ratios.

In contrast, significant changes can be seen in the SSD occupancy ratios among four VMs with the cache-unfriendly workloads, as shown in Fig. 3.8(b). We further observe that VMs (e.g., stg1, web2 and usr2) occupy most SSD resources in $Z_L$ when they experience bursty I/Os. On the other hand, the amount of SSD recourses assigned to VM src21 is less due to its low I/O load. Similarly, under the mixed workload (e.g., MSR−F1U in Fig. 3.8(c)), more SSD resources are assigned to VMs that have relatively larger working set sizes and more I/O requests.

In summary, the results shown in Figs. 3.8(a)-(c) demonstrate that GREM can capture workload changes (in terms of working set sizes and bin accesses) and adapt to these changes by

adjusting the reserved SSD resources in $Z_L$ for each VM based on their bin popularities.

#### 3.1.3.5 Dynamic Partition of $Z_L$ and $Z_S$

Now, we turn to investigate how D_GREM dynamically partitions the overall SSD resources between $Z_L$ and $Z_S$. Figs. 3.8(d)-(f) present the runtime partitioning under D_GREM for three types of `MSR` workloads. We first observe that the capacity of $Z_L$ is much larger than that of $Z_S$. This can be interpreted by observing that $Z_L$, which caches the "always" popular bins, contributes more to the overall I/O hit ratio than $Z_S$. However, we can also see spikes in the zone size of $Z_S$ under the cache-unfriendly workloads (e.g., `MSR-U` in Fig. 3.8(e)). These significantly increasing capacities of $Z_S$ are caused due to the detection of bursty I/Os. D_GREM then aggressively enlarges the capacity of $Z_S$ based on its bursty-case strategy.

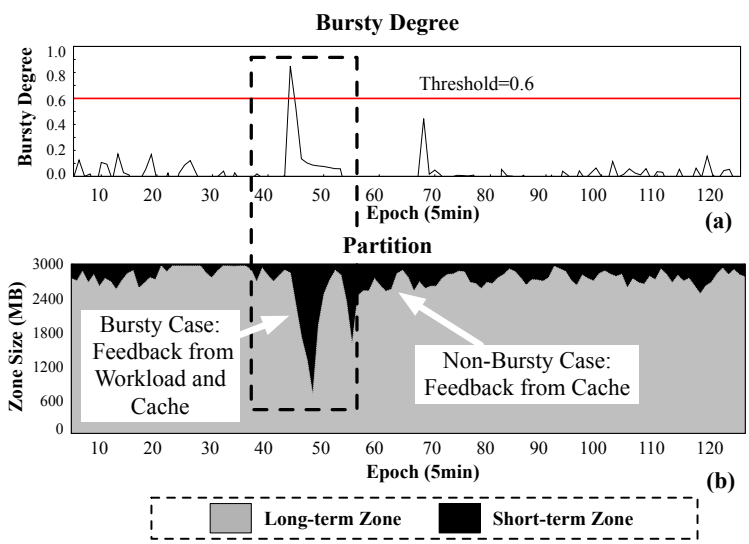



Figure 3.9: Bursty detection of D_GREM.

We then look closely at D_GREM's bursty detector. Fig. 3.9 illustrates a sample piece of D_GREM's bursty detection and reaction under the `MSR-F1U` workload. Here, the bursty degree (i.e., $B_d$ in Eq. 3.3) is in the range of [0,1], and we set the threshold $\beta = 0.6$. We can see that during non-bursty periods, D_GREM finds the best partition based on the importance of two zones (i.e., the contribution to overall I/O hit ratio) such that $Z_L$ has a larger capacity than $Z_S$. When a burst of I/Os is detected, e.g., $B_d$ is beyond the threshold (0.6) at the 44-$th$ epoch in Fig. 3.9(a), D_GREM switches to the bursty case strategy that aggressively increases the size of $Z_S$ based on the amount of accessed bins that have not been cached yet, see Fig. 3.9(b). When the bursty period

ends, D_GREM quickly switches back to the feedback based operation, which can gradually return to the best partitioning for non-bursty cases.

#### 3.1.3.6 Cache Utilization

Recently, a new dynamic and self-adaptive framework is developed called vCacheShare [62], which has a close design goal compared to our work. Therefore, we conduct a set of experiments to compare D_GREM with vCacheShare in terms of I/O hit ratio, I/O cost, and SSD utilization in this subsection. Figs. 3.8(g)-(j) present the runtime SSD resource utilizations (i.e., the percentage of occupied SSD resources) under these two algorithms. Both overall SSD utilizations and per VM's utilizations are depicted in the figure. In these experiments, we have two sets of workloads that are made up with four representative `MSR` workloads, e.g., `mds0`, `src12`, `stg0` and `usr0`. In detail, as vCacheShare is designed to cache data for read I/Os only, we generate the first workload set by manually converting all I/Os of these four workloads to `r`eads for comparing D_GREM with vCacheShare under its best case. We refer this to as "high read ratio" set (i.e., read ratio is 100%), as shown in Fig. 3.8(g) and (h). The second workload set keeps the original read/write ratios (i.e., with the average read ratio of 23.22%) of the four workloads to compare these two algorithms under the general case. We refer this to as "low read ratio" set as shown in Fig. 3.8(i) and (j). The total capacity of SSD resources is 3GB.

First, we can see that both D_GREM and vCacheShare work well under the workload set with high read ratio. The overall SSD utilization quickly reaches to 100%, see Figs. 3.8(g) and (h). We also notice that under vCacheShare, some VMs (e.g., usr0) always dominate the SSD resources. On the other hand, D_GREM leverages both bursty detection and the feedbacks on I/O hits to online assign SSD recourses to each VM. As a result, we observe the fluctuation in each VM's SSD utilizations which accurately reflects the changes in the workload. However, when there is less number of reads in the workload, the effectiveness of vCacheShare diminishes. As shown in Fig. 3.8(i), the SSD resources are not fully utilized until the 350-$th$ epoch under vCacheShare. In contrast, D_GREM considers both write and read I/Os for SSD caching, and reaches 100% SSD utilization immediately after the warming up period. The main reason is that vCacheShare treats SSDs as a read-only cache, and thus bypasses all write I/Os directly to the hard disks. This can also be seen in Fig. 3.8(i), where vCacheShare allocates very few SSD recourses to `src12` (i.e., the blue curve), because `src12` has a very small number of read I/Os.

It is worth mentioning that although vCacheShare also considers handling bursty cases,

| Workload | Hit Ratio Improve (%) | I/O Cost Improve (%) |
|---|---|---|
| High Read Ratio | 15.64 | 34.03 |
| Low Read Ratio | 22.04 | 25.95 |

Table 3.3: Improvement of D_GREM to vCacheShare.

vCacheShare's bursty detection simply relies on one epoch's reuse degree, i.e., $WV/WS$. Meanwhile, D_GREM not only uses the reuse information collected across several consecutive epochs to detect bursty I/Os, but also uses bin popularities, accessed bin amounts and the contributions to I/O hit ratios of each zone during a sliding window to improve both I/O hit ratio and I/O cost. For example, as plotted in the orange curves of Figs. 3.8(h) and (j), VM3 (i.e., stg0) can obtain more SSD resources under D_GREM than under vCacheShare during its non-bursty periods to absorb more short-term hot bins. Therefore, as shown in Table 3.3, D_GREM delivers higher overall I/O hit ratios and lower I/O costs compared with vCacheShare under both workloads with high and low read ratios. Moreover, since D_GREM does not bypass write I/Os, D_GREM has better hit ratio improvement under low read ratio case than high read ratio case.

### 3.1.4 Summary

We present GREM, a new global SSD resource management scheme to allocate a suitable amount of SSDs to heterogeneous VMs. The first design goal is to best utilize the SSD resources by maximizing the I/O hit ratio and minimizing the I/O costs, and the second design goal is to further balance the write cycles in among disks inside the flash-intensive disk pool. GREM splits the entire SSD space into the long-term and short-term zones and takes dynamic I/O demands of all VMs into consideration for reserving SSD resources in the long-term zone to each VM. We further developed D_GREM to dynamically adjust the partition of SSDs between two zone by leveraging the feedback of workload changes and SSD performance. We show that our new schemes allow VMs with different types of workloads to utilize the benefits of SSDs and thus improve the overall I/O hit ratio. We also show that D_GREM successfully detects the changes (or bursts) in I/O workloads and quickly adapts to the changes by shifting SSD resources between two zones. The I/O hit ratio is further improved under D_GREM.

## 3.2 Replication Management for Reliability and Accessibility

### 3.2.1 Motivation

Apart from I/O performance, data reliability is also an important aspect of datacenter storages. Especially with the explosive growth of data creation and access, any loss of, or lack of access to, data would be detrimental to the cloud vendor. In SSD-HDD hybrid datacenters, high-speed SSDs are often utilized as cache tier to accelerate the I/O performance. For example, as shown in the left subfigure of Fig. 3.10, a server node may have multiple storage devices such as SSDs, performance-oriented HDDs, and archive-oriented HDDs as shown in the dashed box. Above that, multiple virtual machines (VMs) running cloud computing applications are hosted by the hypervisor software, and all of them are sharing the storage pools. The right subfigure of Fig. 3.10 further illustrates that in order to speed up the I/O performance of the storage system, SSDs are used to cache hot data, and HDDs are designed to host backend cold data.

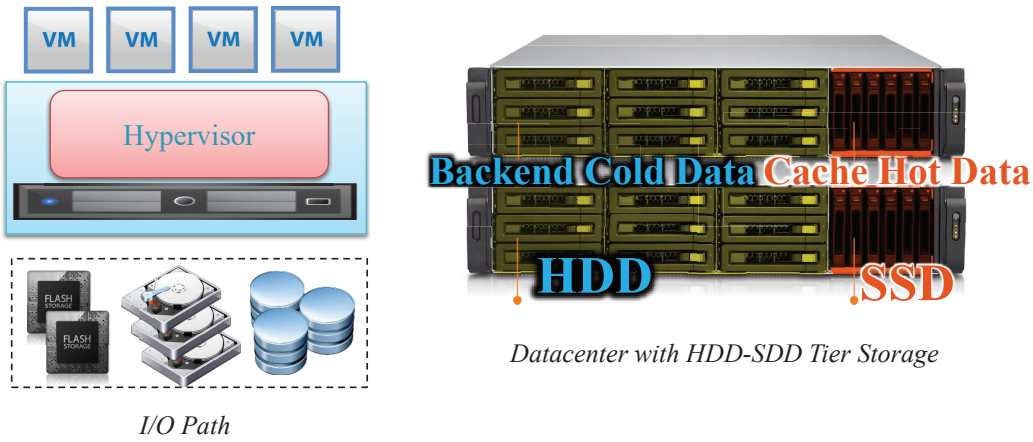



Figure 3.10: Storage architecture of each node.

This multi-tier storage solution may improve the performance of I/Os, but different types of failures may happen as a result of catastrophic events. Data loss and delay caused by disasters will dramatically reduce the data availability and consistency, which are very critical for cloud applications. To address this challenge, replication technique – a process of synchronizing data across multiple storage nodes – is often used to provide redundancy and increase data availability from the loss of a single storage node [141, 142].

However, since redundancy brings overheads in terms of network traffic, I/O performance,

storage space, and consistency maintenance, we need to balance the replication and performance [143, 49]. In practical, SSDs are often used as the *write back* cache to improve to I/O speed, having only one up-to-date copy on SSD is not acceptable for high-SLA demand use cases such as bank, stock market, and military networks. According to the study [13], compare to HDD, SSD is not a "safe destination" though SSD can preserve the data after power off. Therefore, we focus on replicating only cached datasets in the SSDs, and now the main problem is *"where to store replicas of those datasets cached in the SSD while not downgrading the performance?"*

Motivated by this, we propose a complete solution called "AutoReplica", which is an automatic and scalable data replication manager designed for distributed cloud computing infrastructure with SSD-HDD tiering storage systems. AutoReplica maintains replicas of local SSD cache in the remote SSD(s) connected by high-speed fibers, since the access speed of remote SSDs can be faster than that of local HDDs. AutoReplica can automatically build and rebuild the cross-node replica structure following three approaches designed based on different SLAs. AutoReplica can efficiently recover from different disaster scenarios (covers from virtual machine crash, device failures to communication failures) with limited and controllable performance downgrades with a lazy migrate-on-write technique called "fusion cache", which can conduct seamless online migrations to balance loads among nodes, instead of pausing the subsystem and copying the entire dataset from one node to the other. Finally, AutoReplica supports parallel prefetching from both primary node and replica node(s) with a novel dynamic optimizing streaming technique to further improve I/O performance. We implemented AutoReplica on VMware ESXi [17] platform, and experimental results based on the real world I/O workloads show that AutoReplica can significantly improve the performance with slight or even fewer overheads compared to other solutions.

### 3.2.2 Topological Structure

We first introduce the topological structure of the AutoReplica cluster. As illustrated in Fig. 3.11, there are multiple nodes in the cluster, and each node is a physical host which runs multiple virtual machines (VMs). In our prototype, we use VMware's ESXi [17] to host VMs. Inside each node, there are two tiers of storage devices: SSD tier and HDD tier. The former tier is used as the cache and the latter tier is used as the backend storage. Each storage tier contains one or more SSDs or HDDs, respectively. RAID mode disks can also be adopted in each tier. SSD and HDD tiers in each node are shared by VMs and managed by the hypervisor.

Since nodes are connected by high-speed fiber channels, the remote SSD access speed

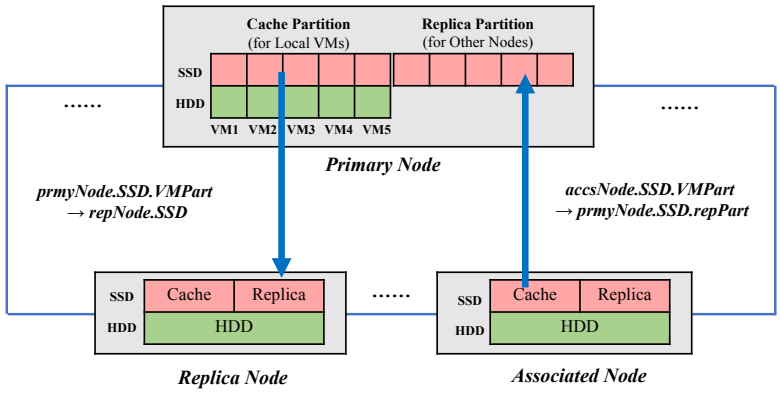
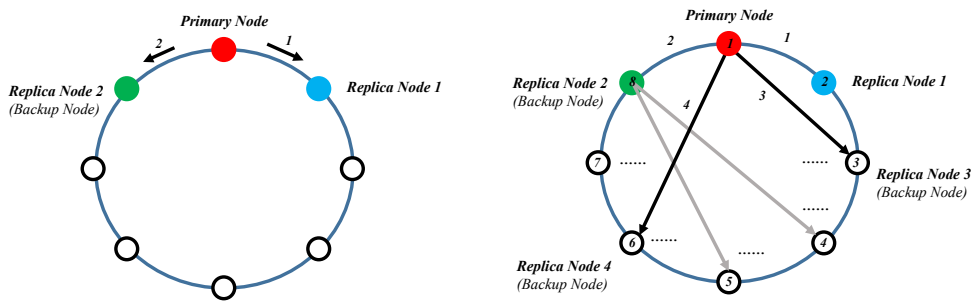



Figure 3.11: An example of the structure of AutoReplica's datacenter.

(including the network delay) can be even faster than local HDD access speed. Thus, to utilize remote SSDs as replica destination, inside the SSD tier, we set two partitions: "Cache Partition" (for the local VMs), and "Replica Partition" (for storing replica datasets from other nodes SSD cache). AutoReplica uses *write back cache policy* to maximize I/O performance, since writing through to HDD will slow down the I/O path. However, as mentioned, SSD is relatively vulnerable and not cannot be equally trusted as a "safe destination" like HDD, though SSD can preserve the data after power off. Therefore, AutoReplica maintains additional replicas in the remote SSDs to prepare for recoveries for failures. In fact, we still can use local HDD as the second replica device for those extremely high SLA nodes, which will be discussed in Sec. 3.2.2.3. Based on these factors, we propose three approaches to setup the topological structure of the datacenter clusters, focusing on *"how to select replica nodes?"*, *"how many replicas nodes do we need?"*, and *"how to assign replicas?"*.

#### 3.2.2.1 Ring Approach

Our first approach is a directed logical "Ring" structure, which can be either user-defined or system-defined. A system-defined ring is based on geographic distance parameters (e.g., I/O latency and network delay). As shown in Fig. 3.12(a), this logical ring defines an order of preference between the primary and replica nodes. Caching is performed by using the local SSD, and the cached data replicated to another node in the cluster. Each node consists of two neighbors (i.e., replica node and associated node as shown in Fig. 3.12(a)), storing replicas on both/one of them. The node walks in the ring until it can find a replica to use if unsuccessful during the process of building the ring




**(a) Ring Structure** **(b) Network Structure**

Figure 3.12: Examples of (a) Ring and (b) Network approaches.

cluster. Once it has a replica, it can begin to write caching independently of what the other nodes are doing.

#### 3.2.2.2 Network Approach

As a "linear" approach, the "Ring" structure has a drawback during searching and building replicas, since it has only one or two directions (e.g., previous and next neighbors). In order to improve the system robustness and flexibility, we further proposed the "network" approach – a symmetric or asymmetric network, see Fig. 3.12(b), which is based on each node's preference ranking list of all its connected nodes (i.e., not limited to two nodes).

| From \ To | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| 1 | - | 1 | 3 | - | - | 4 | - | 2 |
| 2 | ... | - | 1 | ... | 2 | ... | ... | ... |
| 3 | 3 | ... | - | ... | 1 | ... | 2 | ... |
| ... | ... | ... | ... | ... | ... | ... | ... | ... |
| 8 | ... | ... | ... | 2 | ... | ... | 1 | - |

Table 3.4: Example of the "distance matrix" used in *Network* approach, which shows the ranking of each path.

In real implementation, we introduce a "distance matrix" (an example is shown in Table 3.4) to maintain each node's preference list ranked by a customized "score" calculated based on multiple parameters such as network delay, I/O access speed, space/throughput utilization ratio, etc. This

matrix is periodically updated through runtime measurement (e.g., heartbeat). For example, in Table 3.4, node 1's first neighbor is node 2, and its second neighbor is node 8, etc.

The main procedure of how to assign the replica nodes for each node is as following: Each node searches the matrix and selects its "closest" node as its replica node if possible. To avoid the "starvation" case where lots of nodes are choosing one single node or a small range of nodes as their replica nodes, AutoReplica also limits the maximum replica number per node. Lastly, each node can also have more than one replica node.

#### 3.2.2.3 Multiple-SLA Network Approach

In real environment, rather than treating different nodes equally, the system administrator is often required to differentiate users based on their SLAs and workload characteristics. To support this requirement, we further develop the "multiple-SLA network" approach to allow each node to have more than one replica node with different configurations based on a replica configuration decision table.

| Case | Workload | | Destination | | | | #Reps. |
|---|---|---|---|---|---|---|---|
| | SLA | Temp. | $\mathbf{SSD_P}$ | $\mathbf{SSD_{R1}}$ | $\mathbf{SSD_{R2}}$ | $\mathbf{HDD_P}$ | |
| 1 | | ✓ | ✓ | ✓ | | | 1 |
| 2 | | | ✓ | | ✓ | | 1 |
| 3 | ✓ | ✓ | ✓ | ✓ | (✓) | | 1(2) |
| 4 | ✓ | | ✓ | ✓ | | (✓) | 1(2) |

Table 3.5: Replica configuration table for multiple SLAs.

An example of replica configuration table is shown in Table 3.5, where $SSD_P$, $SSD_{R1}$, $SSD_{R2}$ and $HDD_P$ stand for the SSD tier of the primary node, the SSD tier of the first replica node, the SSD tier of the second replica node, and the HDD tier of the primary node, respectively. It also considers:

- **SLA:** Related with importance of each node. Multiple SLAs are supported by utilizing multiple replica configurations. Although our example has only two degrees: "important" and "not important", AutoReplica supports more fine-grained degrees (even online-varying) SLAs.

- **Temperature:** Similar to [140], we use "data temperature" as an indicator to classify data into two categories according to their access frequency: "hot data" has a frequent access pattern, and "cold data" is occasionally queried.

Local HDD ($prmyNode.HDD$) can also be used as a replica destination (i.e., case 4 in Table 3.5), and AutoReplica needs reduce the priorities of those write-to-HDD replica operations in order not to affect those SSD-to-HDD write back and HDD-to-SDD fetch operations in the I/O path.

### 3.2.3 Cache and Replacement Policies

**Main Procedure of AutoReplica's Cache Policy**

**Note:** (1) $prmyNode$ is the primary node. $repNodeRem$ and $repNodeOut$ are the original two replica node. $repNodeIn$ is the destination of migration which replaces $replicaNodeOut$.
(2) $write(inputData, inputDirtyFlag)$: A function that writes the *inputData* into the device. If the device is "SSD", then this function sets *dirtyFlag* of the *inputData* as *True*. If the device is "HDD", then $inputDirtyFlag$ can be ignored.
(3) $migrateTrigger(\,)$: A function returns *True* if the subsystem needs to migrate due to imbalance load.
(4) $T_W$: window size (i.e., frequency) of migration condition checking.

```
Procedure cache (prmyNode, repNode1, repNode2)
    migModeFlag = Flase
    for each new I/O request newData ∈ IOStream on prmyNode do
        /* check load balance */
        if currTime mod T_W == 0 and migModeFlag ≠ True and migTrigger( ) ≠ False then
            migModeFlag = True
            repNodeOut = selectOverheatNode(repNode1, repNode2)
            repNodeIn = selectNextReplica()
            if repNodeIn == repNode1 then
                repNodeRem = repNode2
            else
                repNodeRem = repNode1
        /* online migrate mode cache policy */
        if migModeFlag == True then
            cacheOnlineMigrateMode(newData, prmyNode, repNodeRem, repNodeOut, repNodeIn)
            if repNodeOut.SSD.sizeOfRepForPrmy() == 0 then
                migModeFlag = False
        /* runtime mode cache policy */
        else
            cacheRuntimeMode(newData, prmyNode, repNode1, repNode2)
    return
```

Figure 3.13: Main procedure of AutoReplica's cache policy.

To maximize the I/O performance, AutoReplica uses *write back cache policy*. In detail, when the SSD tier (i.e., cache) is full, SSD-to-HDD eviction operations will be triggered in the

(local) primary node ($prmyNode$); while in the replica node ($repNode$), the corresponding dataset will simply be removed from the $repNode.SSD$ without any additional I/O operations to the $repNode.HDD$. Fig. 3.13 shows a two-replica-node implementation. In fact, it can have any number of SSD replica nodes to support more fine-grained SLAs.

Specifically, AutoReplica's cache and replacement policy is basically switching between two modes, namely "runtime mode" (line 19) and "online migration mode" (line 3 to 11) by periodically checks the $migTrigger$ condition (which considers runtime states such as load balancing and bandwidth utilization). If $migTrigger$ returns true, AutoReplica will select the "$overheat$" replica node (line 6) to be replaced with the next available replica node (line 7). After that, AutoReplica begins to run under the "migration mode" (line 14). If the "migrate out" replica node ($repNodeOut$) has no more "out-of-date" replica datasets (i.e., the migration is finished), AutoReplica then stops the migration by setting $migModeFlag$ to $false$ (line 16), and goes back to the runtime mode (line 19). We describe the details of the runtime mode and the migration mode cache policy in Secs. 3.2.3.1 and 3.2.3.2.

#### 3.2.3.1 Runtime Mode Cache Policy

Under the runtime mode, AutoReplica searches the new I/O request in the local SSD cache partition (i.e., $prmyNode.SSD$). If it returns a cache hit, then AutoReplica either fetches it from the $prmyNode.SSD$ for a read I/O, or updates the new data to its existing cached copies in the $prmyNode.SSD$ and the SSD replica partition in corresponding replica node(s) for a write I/O. For the cache miss case, AutoReplica first selects a victim to evict from the $prmyNode.SSD$ and all the victim's copies from the $repNode.SSD$(s), and then AutoReplica only writes those unsync (with "dirty" flag) evicted datasets into the $prmyNode.HDD$. AutoReplica then inserts the new dataset into both the $prmyNode$ and all its $repNode.SSD$(s). If it is a read I/O, AutoReplica fetches it from the $prmyNode.HDD$ to SSDs of the $prmyNode.HDD$ and also send it to all its $repNode(s)$. It finally returns the fetched $cacheData$ to the user buffer in the memory. If it is a write I/O, AutoReplica simply writes it to SSDs of the $prmyNode$ and all its $repNode$(s) with $dirtyFlag$ as "dirty", since it is unsync new data.

An example is shown in Fig. 3.14, where for the write I/O data "F", AutoReplica first writes back a victim "E" from the $prmyNode.SSD$ to the $prmyNode.HDD$, and deletes "E" from both the $prmyNode.SSD$ and the $reNode.SSD$. Lastly, it writes "F" to both the $prmyNode.SSD$ and the $reNode.SSD$. Our implementation is also compatible for other replacement algorithms to

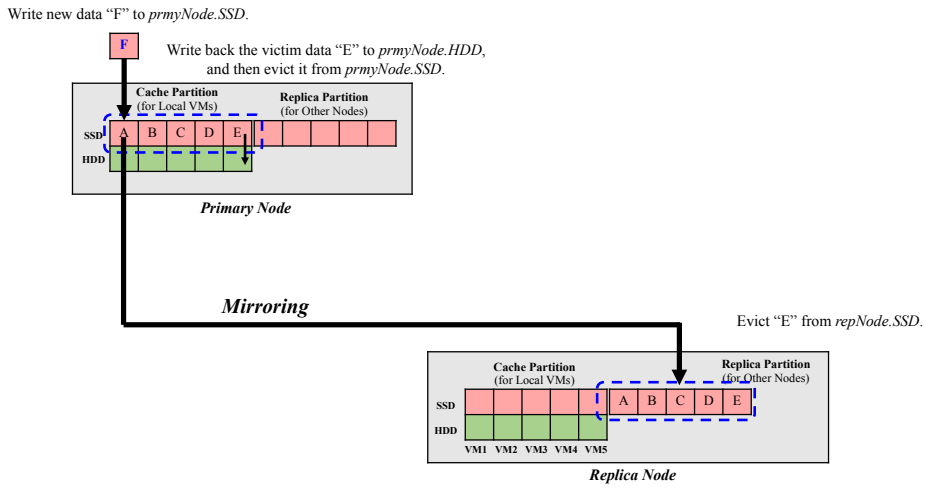



Figure 3.14: Example of runtime cache policy.

implement the victim selection function, such as Glb-VFRM [140] and GREM [20].

#### 3.2.3.2 Online Migration Mode Cache Policy

AutoReplica uses a cost-efficient migrate-on-write scheme called "fusion cache" to migrate replicas from one $repNode$ to the other. The main idea is that instead of pausing the subsystem and copying all existing replicas from the old node ($repNodeOut$) to the new node ($repNodeIn$), regardless of whether these data pieces are necessary or not, "fusion cache" keeps the subsystem alive and only writes new incoming datasets to the $repNodeIn$ and keeps those "unchanged" cached data on the $repNodeOut$. Eventually, the $repNodeIn$ will replace the $repNodeOut$. In other words, AutoReplica mirrors the $prmyNode.SSD$ by using the *unibody* of the $repNodeOut$ and the $repNodeIn$ to save lots of bandwidth.

An example is depicted in Fig. 3.15, where only one replica node is needed to be "migrated out" and one new replica node is needed to take over those cached datasets. When a new write I/O data "F" comes to the $prmyNode$, similarly, AutoReplica first writes back the victim "E" from the $prmyNode.SSD$ to the $prmyNode.HDD$ since the cache is full. Then, "E" on the old replica node is directly deleted. After that, the new write I/O "F" will be inserted to the $prmyNode.SSD$ and its new $repNodeIn.SSD$. As mentioned, the $prmyNode$ (e.g., the blue dash box on top of Fig. 3.15) is mirrored in the "fusion cache" across the old and new $repNodes$ (e.g., the blue dash box in on the right side of Fig. 3.15). Notice that if "F" is a read I/O, AutoReplica will not migrate it from the old to the new replica node.

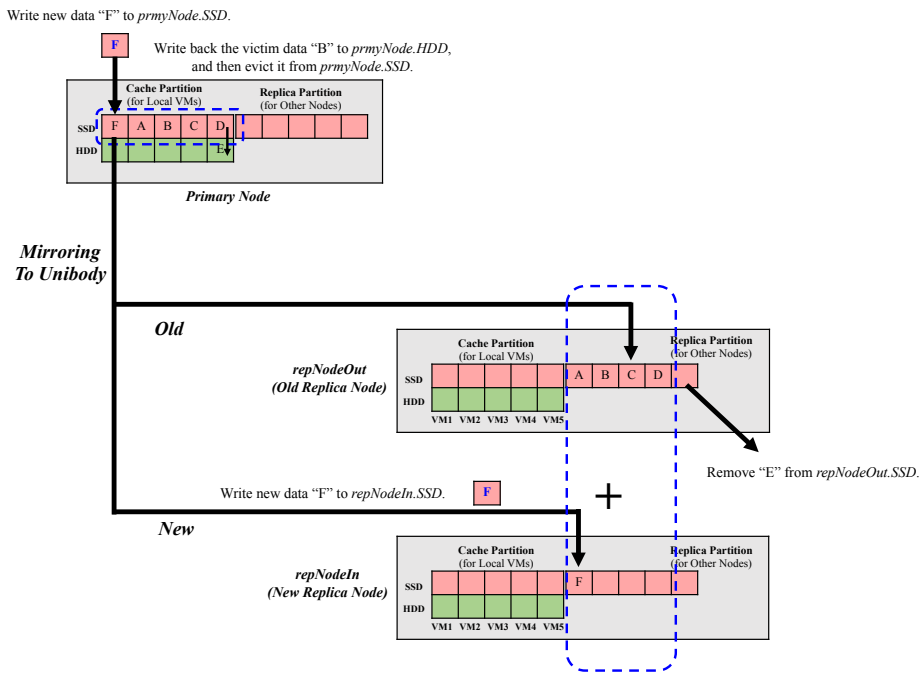



Figure 3.15: Example of online migration cache policy.

To sum up, unlike the traditional pro-active migration, AutoReplica is following this "migrate on write" scheme which is lazy and cost-efficient.

### 3.2.4 Recovery Policy

AutoReplica maintains additional replicas in the remote SSDs to prepare for recoveries for different failures. Specifically, it has different procedures to recover from failures covering the following four scenarios:

#### 3.2.4.1 VM Crash on Primary Node

A very common failure is that a virtual machine crashes on the primary node. As shown Fig. 3.16(1) and Fig. 3.17, AutoReplica first closes out the VMDK. It then writes back "dirty" datasets from the $prmyNode.SSD$ to the $prmyNode.HDD$, and marks all $prmyNode$'s replicas in the $repNode.SSD(s)$ with "nondirty" flag. After that, it restarts crashed VM on the $prmyNode$, and continues to forward incoming I/O requests on both the $prmyNode.SSD$ and the $repNode.SSD$.

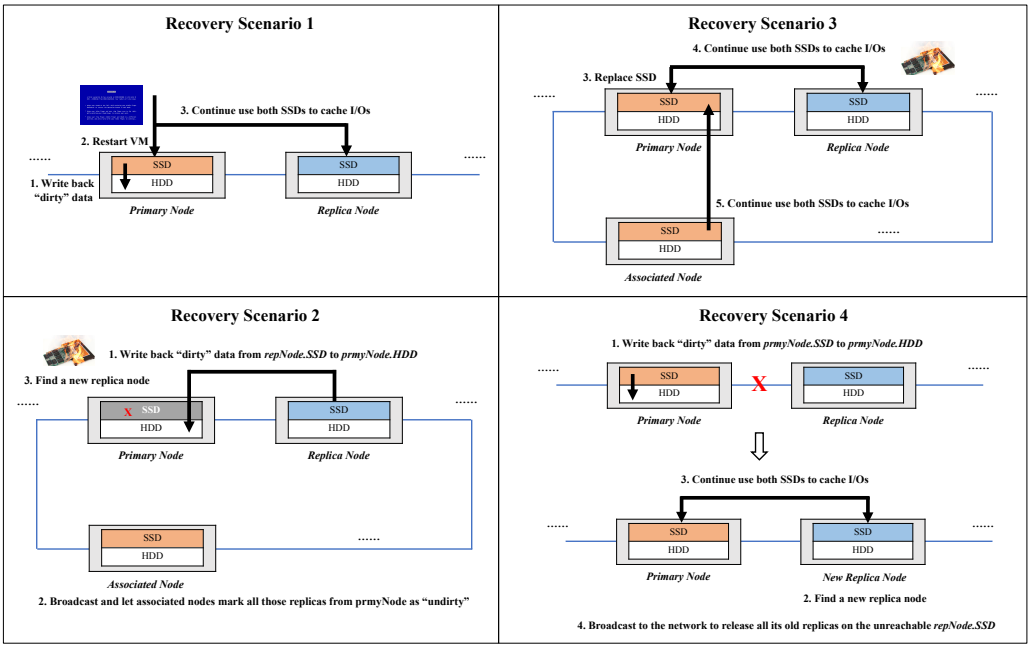



Figure 3.16: Example of AutoReplica's recovery scenarios.

| **[Recovery Scenario 1] VM Crash on Primary Node** | |
|---|---|
| | **Procedure** $recoverFromPrmyNodeVMCrash()$ |
| 1 | Write back "dirty" data from $prmyNode.SSD$ to $prmyNode.HDD$, and mark their copies in $repNode.SSD$ with "nondirty" flag. |
| 2 | Restart crashed VM on $prmyNode$. |
| 3 | Continue to forward incoming I/O requests on both $prmyNode.SSD$ and $repNode.SSD$. |
| 4 | **return** |

Figure 3.17: Main procedure of recovery scenario 1: VM crash on primary node.

#### 3.2.4.2 Primary Node Cache Device Failure

A primary node cache storage device failure will result in its inability to continue to write caching. As shown in Fig. 3.16(2) and Fig. 3.18, AutoReplica first writes back "dirty" datasets from the $repNode.SSD$ to the $prmyNode.HDD$, and marks all $prmyNode.SSD$'s replicas on the $repNode.SSD(s)$ with "nondirty" flag. It then broadcasts this "unavailable" information to notify those nodes having replicas of this failure $prmyNode$ (called "associated nodes") to write back "dirty" datasets from their own SSD to HDD. Notice that replicated datasets with "nondirty" flags are still kept in the $repNode.SSD$(s). AutoReplica further finds and replaces the SSD on the

$prmyNode$. After that, it continues to write incoming I/O requests on both the $prmyNode.SSD$ and the $repNode.SSD$. It also continues to let those "associated nodes" to write new replicas to the $prmyNode.SSD$.

| **[Recovery Scenario 2] Primary Node Cache Device Failure** | |
|---|---|
| | **Procedure** $recoverFromPrmyNodeSSDFailure()$ |
| 1 | Write back "dirty" data from $repNode.SSD$ to $prmyNode.HDD$, and mark their copies in $repNode.SSD$ with "nondirty" flag. |
| 2 | Broadcast this "unavailable" information to the network to let those nodes having replicas in this failure $prmyNode$ ("associated nodes") to write back "dirty" data from their own SSD to HDD, and keep copies with "nondirty" flag in those associated nodes. |
| 3 | Replace SSD on $prmyNode$. |
| 4 | Continue to write incoming I/O requests on both $prmyNode.SSD$ and $repNode.SSD$. |
| 5 | Continue to let those "associated nodes" to write new replicas to $prmyNode.SSD$. |
| 6 | **return** |

Figure 3.18: Main procedure of recovery scenario 2: Primary node cache device failure.

#### 3.2.4.3 Replica Node Cache Device Failure

When a replica node detects a cache device (i.e., SSD) failure, it will disconnect from the primary node and reject any future connection attempts from that node with an error response. As shown in Fig. 3.16(3) and Fig. 3.19, AutoReplica then writes back "dirty" dataset from the $prmyNode.SSD$ to the $prmyNode.HDD$, but still marks and keeps them in the $prmyNode.SSD$ with "nondirty" flag. After a new replica node is found by using dynamic evaluation process, AutoReplica continues to write incoming I/O requests on both the $prmyNode.SSD$ and the new $repNode.SSD$. Notice that policy in Sec. 3.2.4.2 takes responsibility for recovering this failure device.

| **[Recovery Scenario 3] Replica Node Cache Device Failure** | |
|---|---|
| | **Procedure** $recoverFromRepSSDFailure()$ |
| 1 | Write back "dirty" data from $prmyNode.SSD$ to $prmyNode.HDD$, and mark their copies in $prmyNode.SSD$ with "nondirty" flag. |
| 2 | Find a new replica node by using dynamic evaluation process. |
| 3 | Continue to write incoming I/O requests on both $prmyNode.SSD$ and new $repNode.SSD$. |
| 4 | **return** |

Figure 3.19: Main procedure of recovery scenario 3: Replica node cache device failure.

#### 3.2.4.4 Communication Failure Between Primary & Replica Node

When the primary node detects a non-recoverable communication failure between the primary and replica hosts, AutoReplica will be unable to continue to write caching. To recover from this failure, as shown in Fig. 3.16(4) and Fig. 3.20, AutoReplica daemon will write back "dirty" data from the $prmyNode.SSD$ to the $prmyNode.HDD$ to ensure all cached data are updated to the backend HDD. Next, it will start the dynamic evaluation process to find a new replica node to replace the unreachable replica node. It will then continue to use both SSDs to cache I/Os following the "fusion cache" design in migration policy. Finally, it will broadcast to the network to release all its old replicas on the unreachable $repNode.SSD$(s).

| **[Recovery Scenario 4] Communication Failure between Primary and Replica Node** | |
|---|---|
| | **Procedure** $recoverFromCommFailure()$ |
| 1 | Write back "dirty" data from $prmyNode.SSD$ to $prmyNode.HDD$, and mark their copies in $prmyNode.SSD$ with "nondirty" flag. |
| 2 | Find a new replica node by using dynamic evaluation process. |
| 3 | Continue to write incoming I/O requests on both primary SSD and new replica SSD. |
| 4 | Broadcast to the network to release all its old replicas on the unreachable $repNode.SSD$. |
| 5 | **return** |

Figure 3.20: Main procedure of recovery scenario 4: Communication failure between primary and replica node.

### 3.2.5 Parallel Prefetching

Lastly, replicates can also be used to enable parallel prefetching from multiple nodes (similar to parallel stripping in RAID [144]), especially for read operations. An example is shown in Fig. 3.21, where we split the dataset (with the size of $C$) to prefetch (e.g., a file) into two parts (with sizes of $\alpha C$ and $C$), and load each part from the primary and replica node. Assume the access speed of the $prmyNode.SSD$ is $\lambda_1$ (GB/Sec) and the access speed of the $repNode.SSD$ (including the network delay) is $\lambda_2$ (GB/Sec). Since the main target for parallel prefetching is to reduce the total I/O time, i.e., makespan of each I/O request, we can convert our problem into the following optimization

framework:

Minimize:

$$max(\frac{\alpha C}{\lambda_1}, \frac{(1-\alpha)C}{\lambda_2}) \tag{3.8}$$

Subject to:

$$\alpha \in [0, 1] \tag{3.9}$$

$$\lambda_1 \geq \lambda_2 > 0 \tag{3.10}$$

$$\frac{C}{\lambda_1} \geq max(\frac{\alpha C}{\lambda_1}, \frac{(1-\alpha)C}{\lambda_2}) \tag{3.11}$$

Eq. 3.8 is the objective function which minimizes the overall makespan of an I/O request. The makespan is determined by the maximum value of the I/O operating time of each side (i.e., $prmyNode.SSD$ and $repNode.SSD$). Eq. 3.9 ensures that the branching ratio of two streams should be meaningful. Eq. 3.10 reflects that the local SSD I/O speed (i.e., from $prmyNode.SSD$) is usually greater than remote (i.e., $repNode$) I/O speed including network delay. Notice that this constraint can be relaxed as "$\lambda_1 > 0$ and $\lambda_2 > 0$", if the remote I/O speed is higher (which is true in some rare cases), but the optimization framework remains the same. Eq. 3.11 further ensures that the parallel prefetching operation should only be triggered when it *can* help to reduce the I/O makespan.

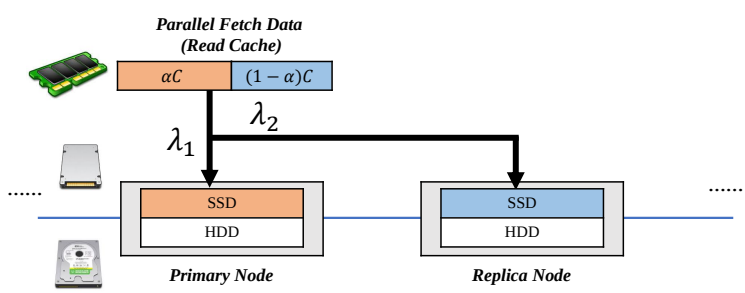



Figure 3.21: Example of parallel prefetch enabled read cache.

Based on this result, Fig. 3.23 further shows the example of the decision maker workflow for parallel prefetch, where the "StatusMonitor" reports to "PararllelPrefectDeamon" and the latter component switches between the "ParallelPrefetchMode" and "LocalPrefetchMode".

We then plot these functions and constraints into Fig. 3.22, where the red line is the objective function curve, and the blue line is the constraint of Eq. 3.11. We can see that there exists a minimum point at the cross point of $f(\alpha) = \frac{(1-\alpha)C}{\lambda_2}$ and $g(\alpha) = \frac{\alpha C}{\lambda_1}$. In order to calculate this sweet

spot, we let:

$$\frac{\alpha C}{\lambda_1} = \frac{(1-\alpha)C}{\lambda_2} \tag{3.12}$$

Then, we can get the minimum makespan ($\frac{\alpha C}{\lambda_1+\lambda_2}$) when:

$$\alpha = \frac{\lambda_1}{\lambda_1+\lambda_2} \tag{3.13}$$

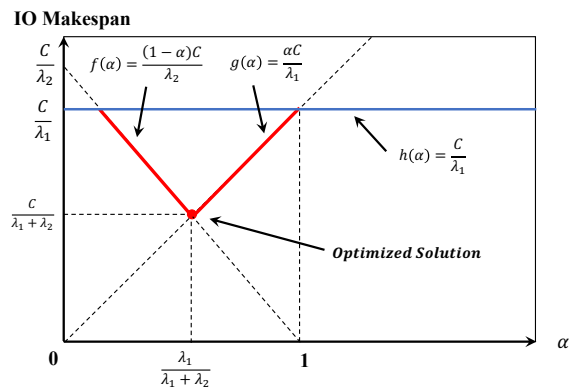



Figure 3.22: Finding the optimized solution of prefetching stream division.

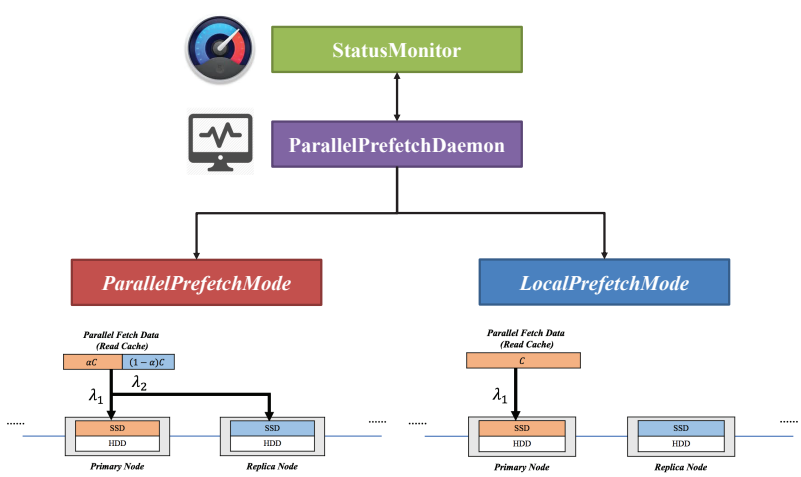



Figure 3.23: Decision maker for parallel prefetch.

We can easily extend it to support any number of replica nodes. Fig. 3.24 first describes the main procedure of parallel fetching daemon, which triggers the parallel prefetching by periodically checking whether the access speed of its all $repNode.SSD(s)$ (including the network delay) is

close enough to the access speed of the local $prmyNode.SSD$ (by comparing their difference with a preset threshold $\varepsilon$), and the current utilization ratio of throughput of the $repNode.SSD(s)$ is less than a threshold $Thr$. Notice that the time window $T_W$ does not necessarily to be same as the sliding window previously mentioned in Fig. 3.13.

**Main Procedure of Parallel Fetching Daemon**

**Note:** (1) $currUtil_{IOPS}$: current throughput utilization rate of a node.
(2) $\epsilon$: preset threshold to compare the difference between local and remote access speed.
(3) $Thr_{IOPS}$: preset threshold of upper bound of throughput utilization rate of a node to be perfected from.

**Procedure** $parallelFetchDaemon()$
1 $parallelReadFlag = False$
2 **while** $True$ **do**
**if** $(currTime \bmod T_W == 0)$ **and** $(\lambda(prmyNode.SSD) - \sum_{i \in RepNodes} \lambda(repNode(i).SSD) \leq \epsilon)$ **and**
3 $(repNode(s).currUtil_{IOPS} \leq Thr_{IOPS})$ **then**
4 $parallelReadFlag = True$
5 **else**
6 $parallelReadFlag = False$
7 **return**

**Parallel Fetching Policy**

**Note:** (1) $\lambda(prmyNode.SSD)$: the IO speed of $prmyNode.SSD$.
(2) $\lambda(repNode(i).SSD)$: the end-to-end IO speed of $repNode(i).SSD$ (including network delay).

**Procedure** $parallelFetch(data)$
1 **if** $parallelReadFlag == True$ **then**
2 fetch size of $\left[\frac{\lambda(prmyNode.SSD)}{\lambda(prmyNode.SSD)+\sum_{i \in RepNodes} \lambda(repNode(i).SSD)} \cdot |data|\right]$ data from $prmyNode.SSD$
3 fetch size of $\left[\frac{\lambda(prmyNode(i).SSD)}{\lambda(prmyNode.SSD)+\sum_{i \in RepNodes} \lambda(repNode(i).SSD)} \cdot |data|\right]$ data from $repNode(i).SSD$
4 **return**

Figure 3.24: Parallel prefetching procedure.

The "ParallelPrefetchMode" will be triggered if all these conditions are satisfied, and the parallel fetching policy then calculates and assigns the branching ratio of the dataset to be loaded from each node. Otherwise, AutoReplica will keep running "LocalPrefetchMode", which only fetches data from the local $prmyNode.SSD$.

### 3.2.6 Evaluation

In this subsection, we investigate the performance of our proposed AutoReplica solution under different use cases.

#### 3.2.6.1 Implementation Configurations

Table 3.6 summarizes the configuration of our server testbed. Fig. 3.25 also illustrates the architecture of our VMware-based implementation.

| Component | Specs |
|---|---|
| Server | PowerEdge R630 Server |
| Processor | Intel(R) Xeon(R) CPU E5-2660 v4 |
| Processor Speed | 2.00GHz |
| Processor Cores | 56 Cores |
| Processor Cache Size | 35M |
| Memory Capacity | 64GB RDIMM |
| Memory Data Rate | 2400 MT/s |
| Operating System | Ubuntu 14.04 LTS |
| Docker Version | 17.03 |
| VM Hypervisor | VMware Workstation 12.5 |

Table 3.6: Configuration of a single server.

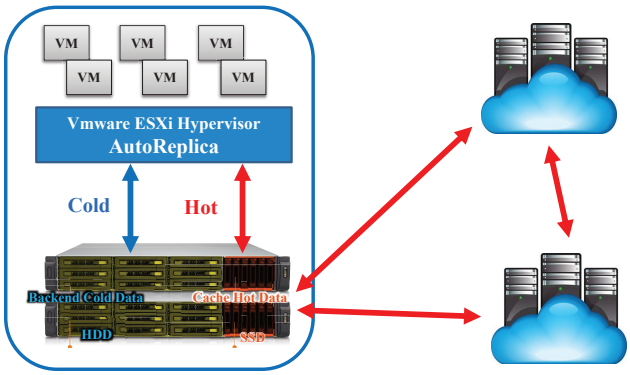



Figure 3.25: Architecture of AutoReplica implementation.

Specifically, there are multiple server nodes in the cluster. Each of these server nodes runs the VMware ESXi hypervisor [17] to host multiple VMs that are working for nearby servers based on location distribution. AutoReplica works on the VMware's ESXi in the "user mode". More accurately, with some internal customization, AutoReplica is very close to a "pseudo-kernel mode" application. AutoReplica consists of 4 components: a vSphere web client plug-in, a CIM

provider, multiple I/O filter library instances and a daemon process on each VM. CPS server nodes are connected by high-speed fiber channels, and due to the advantage of SSD and fiber channels, remote SSDs access speed can be faster than local HDDs speed. AutoReplica strives to cache hot datasets onto the SSD tier and bypass those cold datasets onto the HDD tier.

#### 3.2.6.2 Performance Metrics

In our evaluation, we mainly focus on the following four metrics:

- **Total recovery time:** the cumulative recovery time of all server nodes.
- **Coefficient variation (CV) of total recovery time:** balance degree of the cumulative recovery time across all server nodes. Note that the less the CV is, the better balance is achieved.
- **Overhead:** Total extra I/O and network traffic for recovery.
- **Coefficient variation (CV) of overhead:** balance degree of the overhead across all server nodes.

We use the open source measurement tools (e.g., dstat [145], iostat [146], blktrace [147]) to measure performance metrics such as total recovery time, disk I/O rate, and network traffic. We evaluate the recovery correctness and efficiency of different failure scenarios that are manually triggered and drawn from well-tuned temporal interval distributions. We also evaluate the total recovery time and extra I/O overhead under different cluster sizes. Different SLA configurations are further considered to evaluate the benefit and overhead of multiple replicas.

#### 3.2.6.3 Comparison Solutions

To conduct a fair comparison, we implemented the following three representative solutions for the cluster:

- `NoReplication`: No replications will be generated and maintained in the cluster. Once a server node is failed, data cached in the SSD cannot be recovered from other nodes or local HDD tier. As a result, data re-fetch or re-computation are triggered.
- `Replication(IM)`: A pro-active replication solution, which conducts "Immediate Migration (IM)" on recovery. To be fair, we maintain the same replication assignment as AutoReplica [11] for this solution.

- `AutoReplica(MOW&PF)`: Our proposed AutoReplica solution which has fusion cache with migrate-on-write (MOW), and parallel prefetching (PF) features enabled.

Meanwhile, to consider more scenarios in reality, we also evaluate both homogeneous and heterogeneous VM servers. Details will be discussed in the following two subsections.

#### 3.2.6.4 Study on Homogeneous Clusters

We first investigate the homogeneous cluster use case, which is often referred to as a "symmetrical" structure where same devices are widely deployed in the cluster. This use case is popular due to its simplicity and system consistency [148].

| **clusterSize** | **10** | **20** | **30** | **40** | **50** | **60** | **70** | **80** | **90** | **100** |
|---|---|---|---|---|---|---|---|---|---|---|
| avgRepNode# | 2.20 | 2.45 | 2.63 | 2.33 | 2.62 | 2.45 | 2.41 | 2.39 | 2.43 | 2.45 |
| avgUsrNode# | 1.70 | 2.00 | 2.03 | 1.95 | 2.04 | 1.98 | 2.00 | 1.93 | 1.91 | 1.91 |
| connectRatio(%) | 48.00 | 37.00 | 48.67 | 47.13 | 47.60 | 48.22 | 51.06 | 50.13 | 49.04 | 49.96 |
| avgSSDBW(TB/hour) | 36.00 | 36.00 | 36.00 | 36.00 | 36.00 | 36.00 | 36.00 | 36.00 | 36.00 | 36.00 |
| avgHDDBW(TB/hour) | 0.58 | 0.58 | 0.58 | 0.58 | 0.58 | 0.58 | 0.58 | 0.58 | 0.58 | 0.58 |
| readIO(%) | 30.00 | 30.00 | 30.00 | 30.00 | 30.00 | 30.00 | 30.00 | 30.00 | 30.00 | 30.00 |

Table 3.7: Statistics of configurations of homogeneous clusters.

Table 3.7 shows the statistics of the experiment configurations. In detail, $clusterSize$ is iterated from 10 to 100 server nodes. We use $avgRepNode\#$ and $avgUsrNode\#$ to denote the average number of replication nodes that each node has and the average number of replicated remote node copies that each node is hosting, respectively. Additionally, $connectRatio$ shows that the connectivity of the cluster, which equals the ratio of the number of paths between two nodes to the upper bound of all possible paths (e.g., for $N$-node cluster, at most, we can have $\frac{N^2}{2}$ paths). The larger $connectRatio$ is, the higher chance a node will be recovered from others. We further use $avgSSDBW$ and $avgHDDBW$ to denote the average bandwidth of each node at the SSD tier and the HDD tier, respectively. Finally, $readIO\%$ gives the read I/O ratio of each server node. Notice that for the homogeneous use case, both hardware-related (e.g., $avgSSDBW$ and $avgHDDBW$) and workload-related (e.g., $readIO\%$) factors are the same among different cluster size cases.

Figs. 3.26 to 3.29 depict the results of 10 homogeneous clusters running Cyber-Physical Systems (CPS) workloads [149], where both device failure and communication failure rates are following the uniform distribution. As shown in Fig. 3.26, `NoReplication` has the longest total recovery time, because once a device failure occurs, `NoReplication` has no backups and

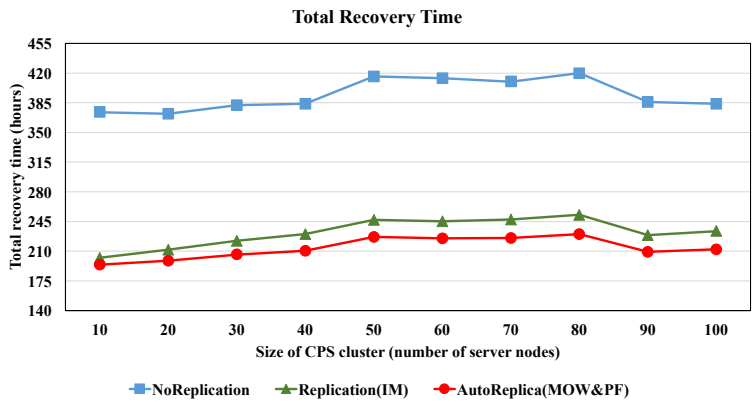



Figure 3.26: Total recovery time under different homogeneous cluster sizes.

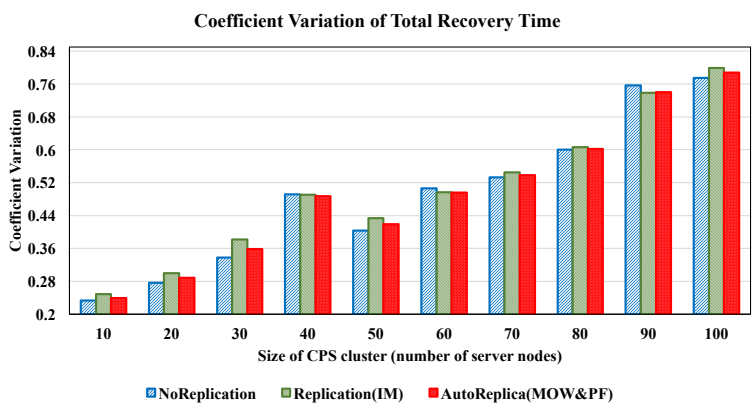



Figure 3.27: Coefficient variation of total recovery time under different homogeneous cluster sizes.

has to request servers to re-fetch or re-compute the lost data. Meanwhile, `AutoReplica` saves 9.28% of the total recovery time from the `Replication` case. The reason is that `AutoReplica` lazily recoveries from its neighbors using the migration-on-write technique and maximizes the I/O bandwidth using the parallel prefetching technique. In contrast, `Replication` has to pause and migrate everything immediately once a device failure or communication failure happens.

We next evaluate the coefficient variation of the total recovery time in Fig. 3.27, where the recovery time of `Replication` has a higher chance to be unbalanced among nodes, while our `AutoReplica` has similar balancing degree to `NoReplication`. This result is promising since in the homogeneous use case, more balanced total recovery distribution helps to make the cluster more robust and stable.

We also investigate the extra I/O and network traffic for the recovery process. As shown

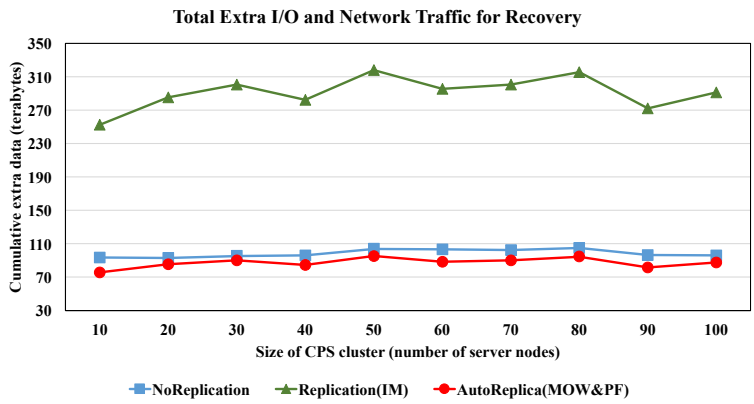



Figure 3.28: Total extra I/O and network traffic for recovery under different homogeneous cluster sizes.

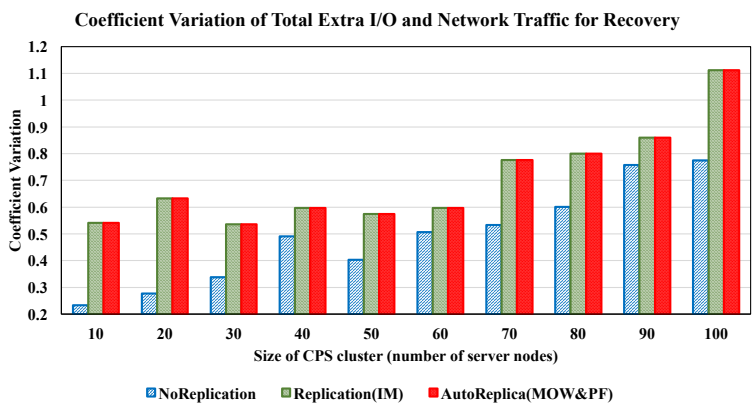



Figure 3.29: Coefficient variation of total extra I/O and network traffic for recovery under different homogeneous cluster sizes.

in Fig. 3.28, the overhead of `Replication` is very large, because it has to migrate everything immediately once a failure happens. Meanwhile, even with the need for maintaining additional replications, `AutoReplica` still has lower overheads than all `NoReplication`'s cases. The main reason behind it is that once a server node failure occurs, `NoReplication` has no backups and has to request servers re-compute and re-send the missing data. These operations trigger a huge amount of extra I/O and network bandwidth consumption.

We further check the coefficient variation of overhead in Fig. 3.29. Neither `Replication` nor `AutoReplica` are load balanced as the `NoReplication` case. Additionally, we observe that although `AutoReplica` saves lots of I/O and network bandwidth, the overhead balancing degree gap between `AutoReplica` and `Replication` cases is very slight. The main reason is

that we deployed the same replication assignment and failure distribution in these two cases in order to conduct a fair comparison.

#### 3.2.6.5 Study on Heterogeneous Clusters

In this subsection, we conduct a set of heterogeneous experiments, where hardware factors (such as $avgSSDBW$ and $avgHDDBW$) and workload factors (such as $readIO\%$) vary among server nodes. Table 3.8 shows the configuration of heterogeneous clusters. Similarly, as we can see from Fig. 3.30, `NoReplication` has the highest total recovery time due to its re-computation cost, and `AutoReplica` still performs better than `Replication` for all cluster sizes. Notice that for the cluster size of 20 and 60, we find the failure interval is relatively larger than that for other cluster sizes. As a result, the scenarios with the cluster size of 20 and 60 have lower chances to have multiple failures happen simultaneously, which then reduces the recovery congestion.

| clusterSize | 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 | 90 | 100 |
|---|---|---|---|---|---|---|---|---|---|---|
| avgRepNode# | 2.40 | 2.45 | 2.77 | 2.58 | 2.60 | 2.45 | 2.39 | 2.38 | 2.44 | 2.58 |
| avgUsrNode# | 1.80 | 1.95 | 2.17 | 2.05 | 2.10 | 1.95 | 1.83 | 1.95 | 1.84 | 2.08 |
| connectRatio(%) | 52.00 | 46.00 | 49.56 | 48.88 | 49.04 | 48.00 | 47.47 | 49.59 | 48.62 | 49.66 |
| avgSSDBW(TB/hour) | 37.60 | 41.15 | 39.27 | 39.13 | 39.34 | 40.72 | 41.04 | 40.73 | 40.68 | 40.71 |
| avgHDDBW(TB/hour) | 0.50 | 0.50 | 0.51 | 0.51 | 0.50 | 0.51 | 0.50 | 0.50 | 0.49 | 0.50 |
| readIO(%) | 35.60 | 36.70 | 37.57 | 37.43 | 38.08 | 36.92 | 38.03 | 36.91 | 37.80 | 37.34 |

Table 3.8: Statistics of configurations of heterogeneous clusters.

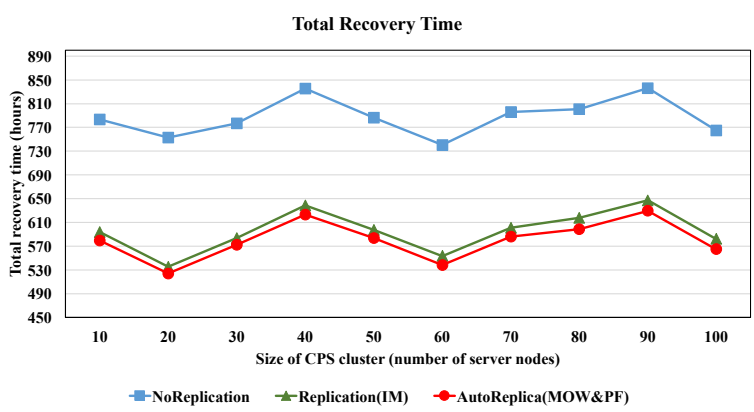



Figure 3.30: Total recovery time under different heterogeneous cluster sizes.

Fig. 3.31 reflects the total recovery time balancing degree. Unlike the homogeneous use case, `Replication` and `AutoReplica` do not have the total recovery time as balanced

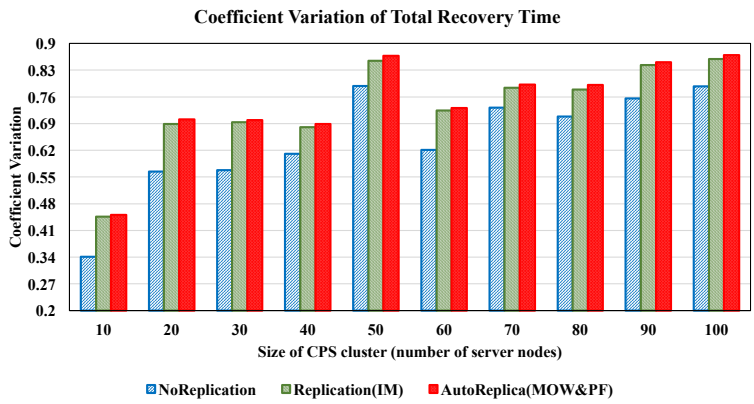



Figure 3.31: Coefficient variation of total recovery time for recovery under different heterogeneous cluster sizes.

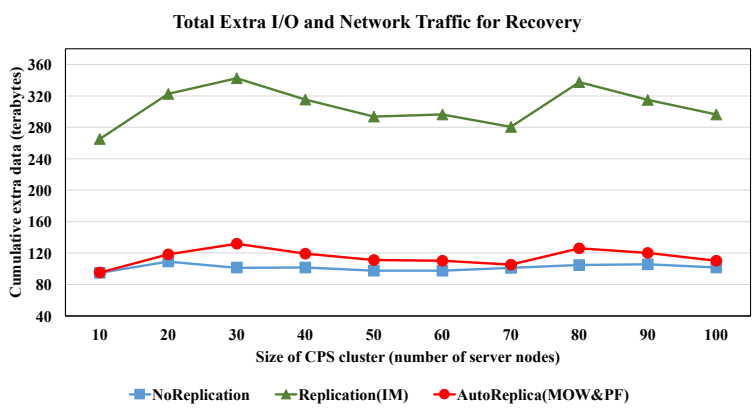



Figure 3.32: Total extra I/O and network traffic for recovery under different heterogeneous cluster sizes.

as `NoReplication` under the heterogeneous use case. The reason is that `NoReplication`'s recovery time is mainly dominated by the data re-computation and re-sending operations. Hence, the recovery time of `NoReplication` is highly coupled with the failure distribution. On the other hand, the recovery time of `Replication` and `AutoReplica` is depending on both failure distribution and replication network assignment.

Similar to the homogeneous use case (e.g., Fig. 3.28), Fig. 3.32 demonstrates that in the heterogeneous use case, `Replication` has the highest overhead. While, `AutoReplica` achieves the shortest total recovery time with a slightly higher overhead compared to `NoReplication`. Fig. 3.33 also illustrates the results of overhead balancing which is similar to the homogeneous case.

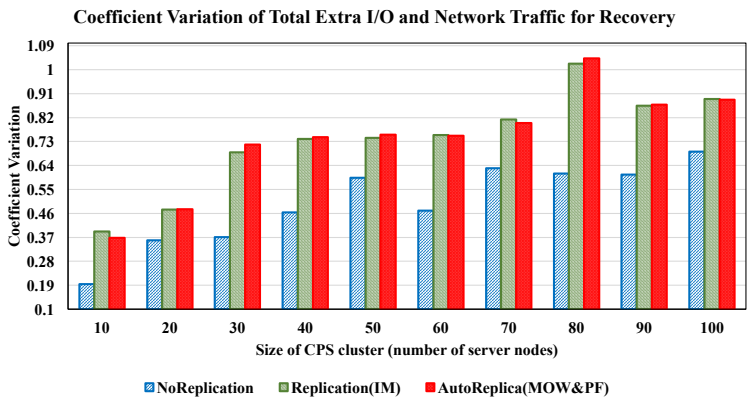



Figure 3.33: Coefficient variation of total extra I/O and network traffic for recovery under different heterogeneous cluster sizes.

### 3.2.7 Summary

We propose a data replica manager solution called "AutoReplica", working in distributed caching and data processing systems using SSD-HDD tier storages. AutoReplica balances the trade-off between the performance and fault tolerance by storing caches in replica nodes' SSDs. It has three approaches to build the replica cluster in order to support multiple SLAs, based on an abstract "distance matrix" which considers preset priorities, workload temperature, network delay, storage access latency, etc. AutoReplica can automatically balance loads among nodes. It also conducts seamless online migration operations (i.e., migrate-on-write scheme), instead of pausing the subsystem and copying the entire dataset from one node to the other. AutoReplica also supports parallel prefetching from both primary node and replica node(s) with a new dynamic optimizing streaming technique to improve I/O performance.

# Chapter 4

# All-Flash Datacenter Storage Systems

Today, the price of flash memory keeps falling and storage capacity of SSDs keeps increasing. As a result, it is widely believed that all-flash storage systems will be adopted in the enterprise datacenters in the near future. In this chapter, we discuss the resource management problems in all-flash datacenter storage systems.

We first investigate how to improve I/O performance for all-flash and multi-tier datacenter storage systems. We find that existing caching and tiering algorithms are not suitable for these all-flash datacenters because they do not consider different specialties (such as high performance, high capacity, etc.) of each SSD tier. Thus, we develop an automatic data placement manager, called "AutoTiering" (Sec. 4.1), to handle VMDK allocation and migration for best utilizing the storage resources, optimizing the performance, and reducing the migration overhead. AutoTiering is based on an optimization framework. The core technique of this framework is to predict VM's performance change on different SSD tiers (which have different specialties) without conducting real migration.

We next focus on how to reduce the total cost of ownership of all-flash datacenter storage systems due to the fact that flash devices with limited write cycles suffer from write amplification. In Sec. 4.2, we develop a TCO model to estimate both the capital and operating expenditures of all-flash storage systems, and then integrate a write amplification model of NVMe SSDs that we build based on empirical data with our this TCO model to reflect flash device endurance. We finally design a new dispatcher called "MINTCO" to guide datacenter managers to make workload allocation decisions under the consideration of minimizing TCO and balancing device endurance.

## 4.1 Automatic Tiering to Harness All-Flash Storages

### 4.1.1 Motivation

To solve the problem caused by I/O bottlenecks, parallel I/O to multiple HDDs in Redundant Array of Independent Disks (RAID) becomes a common approach. However, the performance improvement from RAID is still limited, therefore, lots of big data applications strive to store intermediate data to memory (e.g., RAM) as much as possible such as Apache Spark. Unfortunately, memory is too expensive, and its capacity is very limited (e.g., 64∼128GB per server), so it alone is not able to support super-scale cloud computing use cases. As a device between RAM and HDD, SSD has an acceptable price and space. Since 2008, SSDs started to be adopted in the server market and broke the asymmetric R/W IOPS barrier dramatically, and became one of the promising solutions to speedup storage systems as a cache or a fast tier for slow HDDs [10]. However, as time goes by, SSD-HDD based solutions are no longer competent to meet the current big data requirements, due to the huge I/O speed gap between SSDs and HDDs. On the other hand, the capital expenditure of flash-based SSDs keeps decreasing and the storage capacity of SSDs keeps increasing. Thus, it is widely believed that SSD-HDD solution is just for a transition period, and all-flash multi-tier storage systems will be adopted in the enterprise datacenter in the near future, similar to what happened to HDD-tape solution 30 years ago. For example, high-end NVMe SSDs can replace SATA SSD, and low-end TLC SSDs will replace HDDs.

However, traditional caching algorithms are deemed useful only when the performance difference between storage devices is at least 10x, while the gap between SSD tiers in all-flash datacenters is not that big. More importantly, SSDs are expensive, and it is costly to maintain duplicated copies (one for cache and one for backend data) in two SSD tiers. Thus, we develop an automatic data placement manager called "AutoTiering" to handle VM allocation and migration in an all-flash multi-tier datacenter. The ultimate goal is to best utilize the storage resource, optimize the performance, and reduce the migration overhead, by associating VMs with an appropriate tier of storage. AutoTiering is based on an optimization framework to provide the best *global* (i.e., for all VMs in the datacenter) migration and allocation solution over runtime. AutoTiering's approximation approach further solves the simplified problem in a polynomial time, which considers both historical and predicted performance factors, as well as estimated migration cost. This comprehensive methodology prevents to frequently migrate back and forth VMs between tiers due to I/O spikes. Specifically, AutoTiering uses a micro-benchmark-based sensitivity calibration and regression session to predict

VM's performance change on different tiers without performing real migrations, since different VMs may have different benefits of being upgraded to a high-end tier. We implement AutoTiering on VMware ESXi [17] and evaluate its performance with a set of representative applications. The experimental results show that AutoTiering can significantly improve the I/O performance by increasing I/O throughput and bandwidth as well as reducing I/O latency.

### 4.1.2 Problem Formulation

In this subsection, we first formulate the problem of VM allocation and migration in an all-flash multi-tier datacenter, and then develop an optimization framework to best utilize SSD resource among VMs in the datacenter.

#### 4.1.2.1 System Architecture

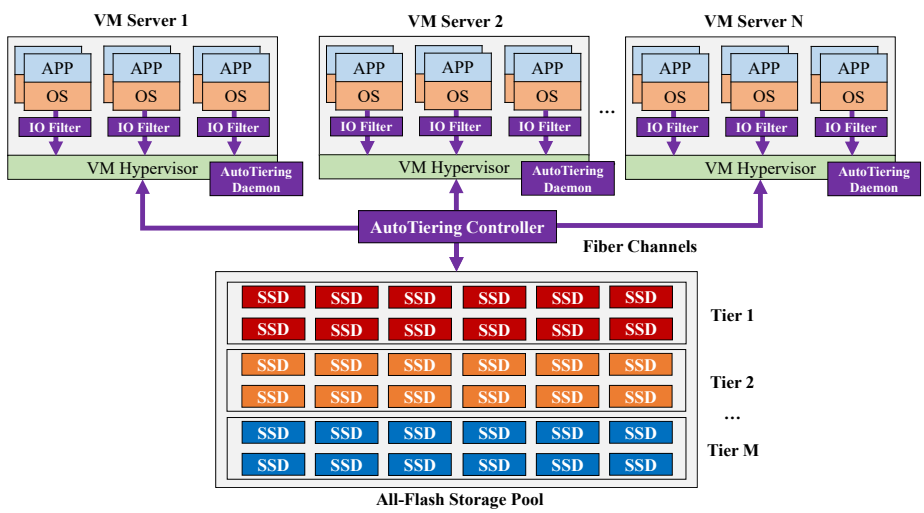



Figure 4.1: Architecture of AutoTiering in a multi-tier all-flash storage system.

As illustrated in Fig. 4.1, AutoTiering has the following components:

- *IO Filter:* Attached to each VMDK (virtual machine disk file) being managed on each host, it is responsible for collecting I/O related statistics as well as running special latency tests on every VMDK. The data will be collected and sent to the AutoTiering Daemon on the host [150].
- *Daemon*: Running on the VM hypervisor of all hosts, it tracks the workload change (i.e., I/O access pattern change) of each VM, collects the results of latency injection tests from the IO Filter, and sends them to the Controller.

- *Controller:* Running on a dedicated server, it is responsible for making decisions to trigger migration based on the predicted VM performance (if it is migrated to other tiers) and the corresponding migration overhead.

#### 4.1.2.2 Optimization Framework

To develop an optimization framework aimed at minimizing the total amount of server hours by determining a VM migration schedule, we formulate the problem by investigating the following factors: First, from each VM's point of view, the reason for a certain VM to be migrated from one tier to another is that the VM can perform better after migration, e.g., less average I/O latency, higher IOPS, higher throughput, etc. Second, the corresponding migration cost need to be considered, because migration is relatively expensive (consumes resource and time) and not negligible. Third, from the *global optimization*'s point of view, it is hard to satisfy all VMs to be migrated to their favorite tiers at the same time due to resource constraints and their corresponding SLAs. Fourth, the *global optimization* should consider overtime changes of all VMs as well as post-effects of migration. For example, the current best allocation solution may lead to a bad situation for the future since VMs are changing behaviors during runtime.

Based on these factors, our optimization framework needs to consider potential benefits and penalties, migration overhead, historical and predicted performance of VMs on each tier, and SLA, as shown in Eq. 4.1 to 4.5. Table 4.1 represents notations that we use in this subsection.

**Maximize:**

$$\sum_{\forall v, \forall t, \forall \tau} w_{v,\tau} \cdot [\sum_{\forall k} \alpha_k \cdot r(v, t_{v,\tau}, \tau, k) - \beta \cdot g(v, t_{v,\tau-1}, t_{v,\tau})], \tag{4.1}$$

**Subject to:**

for $\forall v, \forall \tau$ :

$$t_{v,\tau} \neq \emptyset, \text{and } |t_{v,\tau}| \geq 1, \tag{4.2}$$

for $\forall \tau_1, \tau_2 \in [0, +\infty)$ :

$$r(v, t_{v,\tau_1}, \tau_1, k_s) \equiv r(v, t_{v,\tau_2}, \tau_2, k_s) \geq 0, \tag{4.3}$$

for $\forall v, \forall t, \forall \tau$ :

$$r(v, t_{v,\tau}, \tau, k) = r_{Prd}(v, t_{v,\tau-1}, t_{v,\tau}, \tau - 1, k) \geq 0, \tag{4.4}$$

$$\sum_{\forall v} r(v, t_{v,\tau}, \tau, k) \leq \Gamma_k \cdot R(t_{v,\tau}, k), \tag{4.5}$$

The main idea is to maximize the "*Profit*", which is the entire performance gain minus the performance penalty, i.e., "*Performance Gain - Performance Penalty*", as shown in the objective function Eq. 4.1. The inner "sum" operator conducts a weighted sum of the usage of all types of resources of each VM (e.g., throughput, bandwidth, storage size, etc.), assuming migrating VM $v$ from tier $t_{\tau-1}$ to $t_{\tau}$. The outer "sum" operator further iterates all possible migration cases, where weight parameter $w_{v,\tau}$ reflects the SLA of VM $v$ in $\tau$ epoch. Notice that the term "migration" here is not to migrate a VM from one host server to another. Instead, only backend VMDK files are migrated from one to the other SSD tier. As a result, non-disk-I/O related resources (e.g., host-side CPU and memory) are not considered in this dissertation. Eq. 4.2 guarantees that each VM is hosted by at least one disk tier. In fact, each VM can have multiple VMDKs, and each of them can be located at different tiers. Unlike the previous SSD-HDD tiering work [61, 62] that are operating on fine-grained data blocks due to the HDD speed bottleneck, the minimal unit to migrate here is the entire VMDK of each VM. Eq. 4.3 ensures that storage size (i.e., VMDK size) will not change before and after migrations, where $k_s$ is the type index of storage resource. Eq. 4.4 shows that a prediction model function is utilized to predict the performance gain (for details, see Sec. 4.1.3.2). Eq. 4.5 lists resource constraints, where $\Gamma_k$ is preset upper bound (in percentage) of each type (i.e., $k$-th) of resource that can be used. Finally, the temporal migration overhead is the size of the VM to be migrated, divided by the bottleneck of the migrate-out read speed and the migrate-in write speed, i.e., $g(v, t_{v,\tau-1}, t_{v,\tau}) = \frac{r(v,t_{v,\tau-1},\tau-1,k_s)}{\mu_(v,\tau-1)}$. The migration speed is $\mu_(v, \tau-1) = min([Pr(\Lambda, t_{v,\tau-1}, \tau-1) + Pw(\Lambda, t_{v,\tau-1}, \tau-1))$, where $Pr(\Lambda, t_{v,\tau-1}, \tau-1)$ is the function of available remaining read throughput. Since we are going to *live migrate* VM $v$, the read throughput used by this VM ($Pr(v, t_{v,\tau-1}, \tau-1)$) is also available and has been added back here. $Pw(\Lambda, t_{v,\tau-1}, \tau-1)$ gives the migrate-in write throughput.

Since the system has no information on the future workload I/O patterns, it is impossible to conduct the *global* optimization for all $\tau$ time periods during runtime. Moreover, the decision making process for each migration epoch in the *global* optimal solution depends on the past and future epochs, which means that Eq. 4.1 cannot be solved by traditional sub-optimal-based dynamic programming techniques. Lastly, depending on the complexity of the performance prediction model (e.g., Eq. 4.4), the optimization problem can easily become NP-hard.

| **Notation** | **Meaning** |
|---|---|
| $v$, $v_i$ | VM $v$, $i$-th VM, $v \in [1, v_{max}]$, where $v_{max}$ is the last VM. |
| $t$, $t_i$ | Tier $t$, $i$-th tier, $t \in [1, t_{max}]$, where $t_{max}$ is the last tier. |
| $t_{v,\tau}$ | VM $v$'s hosting tier during epoch $\tau$. |
| $k$ | Different types of resources, $k \in [1, k_{max}]$, such as throughput, bandwidth, storage size, etc. |
| $\tau$ | Temporal epoch ID, where $\tau \in [0, +\infty)$. |
| $\alpha_k$, $\beta$ | $k$-th resource's weight, migration cost weight. |
| $r(v, t_{v,\tau}, \tau, k)$ | Predicted type of $k$ resource usage of VM $v$ running on tier $t_{v,\tau}$. |
| $g(v, t_{v,\tau-1}, t_{v,\tau})$ | Migration cost of VM $v$ during epoch $\tau$ from tier $t_{v,\tau-1}$ to tier $t_{v,\tau}$. |
| $k_s$ | "Storage" resource's index. |
| $\mu(v, \tau - 1)$ | Estimated migration speed of VM $v$ at time $\tau - 1$ epoch. |
| $Pr(v, t, \tau)$, $Pw(v, t, \tau)$ | Read/write resource of VM $v$ on tier $t$ during epoch $\tau$. |
| $Pr(\Lambda, t, \tau)$, $Pw(\Lambda, t, \tau)$ | All remaining available read/write resource of tier $t$ during epoch $\tau$. |
| $\Gamma_k$ | Upper bound (in percentage) of each type of resource that can be used. |
| $R(t_{v,\tau}, k)$ | Total capacity of $k$-th type of resource on tier $t_{v,\tau}$. |
| $L_t$ | Original average I/O latency (without injected latency) of tier $t$. |
| $b_v$, $m_v$ | Parameters of TSSCS liner regression model ($y = mx + b$). |
| $s_v$ | Average I/O size of VM $v$. |
| $S_v$, VM[v].size | Storage size of VM $v$. |
| $w_{v,\tau}$, $wetP[t]$, $wetB[t]$, $wetS[t]$ | Weight of VM $v$ and weights of tier $t$'s each type of resource. |
| $maxP[t]$, $maxB[t]$, $maxS[t]$, $spc[t].P$, $spc[t].B$, $spc[t].S$ | Preset available resource caps and specialties of tier $t$, P=throughput, B=bandwidth, S=storage size. |

Table 4.1: Descriptions of notations used in this subsection.

### 4.1.3 Approximation Algorithm Design

To obtain the result close to the optimized solution in a polynomial time, we have to relax some constraints. In detail, we first downgrade the goal from "*global optimizing for all time*" to "*only optimizing for each epoch*" (i.e., runtime greedy). Furthermore, since we have the foreknowledge of each tier's performance "specialty" (such as high throughput, high bandwidth, large space, small write amplification function, large over-provisioning ratio, large program/erase cycles, etc.), we can make migration decisions based on the ranking of the estimated performance of each VM on each tier, considering each tier's specialties and corresponding estimation of migration overhead. Details of our approximation algorithm are discussed in the following subsections.

#### 4.1.3.1 Main Procedure

Alg. 3 lines 1-8 show the main procedure of AutoTiering, which periodically monitors the performance and checks whether a VM needs to be migrated. Specifically, $monitorEpoch$ is the frequency of evaluating and regressing the performance estimation model, and $migrationEpoch$ is the frequency of triggering VM migration from one tier to the other one. The migration decision is made at the beginning of each $migrationEpoch$, which is greater than $monitorEpoch$. Apparently, the smaller temporal window sizes, the more frequent monitoring, measurement, and migration. The system administrator can balance a tradeoff between the accuracy and the migration cost by conducting a sensitivity analysis before deployment. As shown in line 4 of Alg. 3, procedure $tierSpdSenCalibrate$ estimates VM's performance on each tier based on the regression model. Line 5, thus, calculates performance matrices, and line 6 calculates the score by considering the historical and current performance of each VM, estimates the corresponding migration overhead. Finally, VM migrations are triggered, see line 8. We describe their details in the following subsections.

#### 4.1.3.2 Tier Speed Sensitivity Calibration

In order to estimate each VM's performance on other tiers without conducting any actual migration, we first try to "emulate" the speed of tiers by manually injecting a synthetic latency to each VM's I/Os, and measure the resultant effect on total I/O latency by calling IOFilter APIs (e.g., VMware vSphere APIs for I/O Filtering). A representative result of our preliminary experiment is shown in Fig. 4.2, where we observe that the performance variation can be modeled to a *linear* function. VMs running different types of applications have varying performance sensitivity to the tier. More precisely, a VM with synchronous applications is more sensitive to the tier speed change

**Algorithm 3:** AutoTiering Procedure Part I.

```
1  Procedure autoTiering()
2      while True do
3          if currTime MOD monitorEpoch = 0 then
4              tierSpdSenCalibration();
5              calCapacityMatrices();
6              calScore();
7          if currTime MOD migrationEpoch = 0 then
8              triggerMigration();
9  Procedure tierSpdSenCalibration()
10     for t ∈ tierSet do
11         for v ∈ VMSet[t] do
12             for samplesWithLatency ∈ sampleSet[t][v] do
13                 CV+ = calCV(samplesWithLatency);
14                 samplesWithLatency = avg(samplesWithLatency);
15             CV/ = len(sampleSet[t][v]);
16             if CV ≥ 1 then
17                 VM[v].conf = 0.05;
18             else
19                 VM[v].conf = 1 − CV;
20             (VM[v].m, VM[v].b) = regress(sampleSet[t][v]);
21     return;
22 Procedure calCapacityMatrices()
23     for t ∈ tierSet do
24         for v ∈ VMSet[t] do
25             Lat = estimateAvgLat(v, t);
26             if Lat > 0 then
27                 IOPS = 10^6/Lat;
28             else
29                 IOPS = 0;
30             VMCapMat[t][v].P = IOPS;
31             VMCapMat[t][v].B = (IOPS×VM[v].avgIOSize)/10^6;
32             VMCapMat[t][v].S = VM[v].size;
33     return;
34 Procedure estimateAvgLat(v,t)
35     return VM[v].m × (tierLatency[t] − tierLatency[VM[v].tier]) + VM[v].b;
```

than a VM with asynchronous applications. The main reason is that the latter one needs to wait until each I/O has been completed.

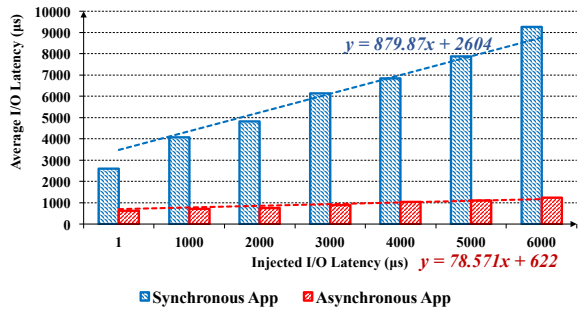



Figure 4.2: Example of VM's performance changes under different injected latencies.

Motivated by this observation, we introduce a micro-benchmark session, called "Tier Speed Sensitivity Calibration Session (`TSSCS`)", to predict (i.e., without conducting actual migration) how much performance benefit (resp. performance penalty) it can take for each VM if we migrate that VM to a faster (resp. slower) tier. In detail, `TSSCS` has the following properties:

- *Lightweight*: Running inside the IOFilter, `TSSCS` injects a synthetic tiny latency to each VM in a very low frequency, without affecting the current hosting workload performance.
- *Multi-Samples per Latency*: `TSSCS` improves the accuracy of emulating each VM's performance under each tier by taking the average over multiple samples that are obtained with the same injected latency of each tier.
- *Multi-Latencies per Session*: `TSSCS` takes multiple latencies per session to refine the regression.
- *Multi-Sessions during Runtime*: `TSSCS` is periodically triggered to update the regression model by calling "$tierSpdSenCalibration$" in Alg. 3 line 4.

Fig. 4.3 depicts an example of three VMs running on three different tiers: VM $v_1$ on tier $t_1$, VM $v_2$ on tier $t_2$, and VM $v_3$ on tier $t_3$. Assume $t_1$ is 2,000 $\mu s$ faster than $t_2$, and $t_2$ is 2,000 $\mu s$ faster faster than $t_3$. We run `TSSCS` on each VM on their hosting tier, and get the latency curves shown in three plots in Fig. 4.3. Since all injected latencies are additional to the original bare latency (i.e., without injected latency), we have to align them according to the absolute latency values (i.e., bare latency + injected latency). Notice that since we cannot inject negative latencies (i.e., obviously we can only slow down the tier), the dash lines in subfigures "VM2 on tier2" and "VM3 on tier3"

are regressed based on the solid lines. After that, we can draw three (colored) lines for each tier based on their absolute latency values (i.e., red for tier1, green for tier2, and blue for tier3). Then, we can easily predict the average I/O latency values of each VM on each tier (i.e., red points for $t_1$, green points for $t_2$, and blue points for $t_3$). We see that VM1 is the most sensitive one to tier speed changes (i.e., with the greatest gradient), while VM3 is the least sensitive one (i.e., relatively flat). Therefore, intuitively, we should assign VM1 to tier1 (fastest tier) and VM3 to tier3 (slowest tier). Alg. 3 lines 9-21 describe the procedure of tier speed sensitivity calibration session (`TSSCS`). In addition, AutoTiering calculates the coefficient variation (CV) of sampling results to decide the estimation confidence, see in lines 13 and 16-19.

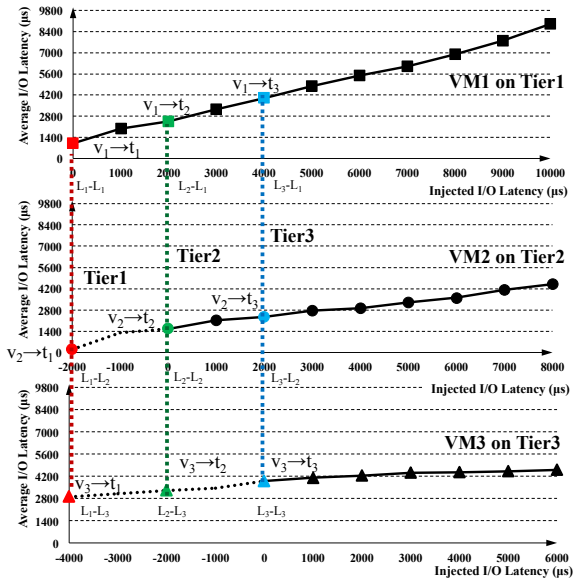



Figure 4.3: Example of average I/O latency estimation.

**4.1.3.2.1 Performance Capacity Matrices** Once we have the average latency vs. injected latency curves of each VM of the current moment, we calculate corresponding performance the estimation of throughput (denoted as $P$, unit in IOPS), bandwidth (denoted as $B$, unit in MBPS), and storage size (denoted as $S$, unit in bytes), and record them into three two-dimensional matrices, i.e., $VMCapMat[t][v].P$, $VMCapMat[t][v].B$, and $VMCapMat[t][v].S$, where $t$ and $v$ are IDs of tier and VM, respectively. Compared to the first two matrices, the last matrix is relatively straightforward to be obtained by calling the hypervisor APIs to measure the storage size that each VM is occupying. As shown in Alg. 3 lines 22-33, AutoTiering updates the VM capacity matrices (i.e., $VMCapMat$) by reiterating for all tiers and VMs. It estimates the "new" latency under other tiers

**Algorithm 4:** AutoTiering Procedure Part II.

```
1  Procedure calScore()
2      for t ∈ tierSet do
3          for v ∈ VMSet[t] do
4              if maxP[t] < VMCapMat[t][v].P or
                 maxB[t] < VMCapMat[t][v].B or maxS[t] < VMCapMat[t][v].S
                 then
5                  VMCapRatMat[t][v].P = 0;
6                  VMCapRatMat[t][v].B = 0;
7                  VMCapRatMat[t][v].S = 0;
8                  tierVMPerfScore[t][v] = −1;
9                  continue;
10             /* Convert to percent capacities */;
11             VMCapRatMat[t][v].P = VMCapMat[t][v].P / maxI[t];
12             VMCapRatMat[t][v].B = VMCapMat[t][v].B / maxP[t];
13             VMCapRatMat[t][v].S = VMCapMat[t][v].S / maxS[t];
14             tierVMPerfScore[t][v] = agingFactor × ttlVMCapRatMat[t][v] +
                 currCapScore(t, v) − wetMig[t] × migCost(t, v);
15     return;
16 Procedure migCost (t,v)
17     migSpd = min(remReadThrput(VM[v].tier) +
         VM[v].currReadThrpt, remWriteThrput(t));
18     return VM[v].size/migSpd;
19 Procedure triggerMigration()
20     for t ∈ tierSet do
21         for v ∈ descendingSortByScore(tierVMPerfScore[t]) do
22             if VM[v].isAssigned = False and tierVMPerfScore[t][v] ≠ −1
                 and tierHasCapacityForVM(t, v) then
23                 assignVMToTier(v,t);
24                 VM[v].isAssigned=True;
25     return;
```

by calling the "$estimateAvgLat(v, t)$" function in Alg. 3 line 25. Lines 34-35 show the detail of $estimateAvgLat$ function, where the input parameters are VM $v$ and target tier $t$ for estimation, which returns an estimation based on the linear regressions of $m$ and $b$ values. Once the estimated average I/O latency results are obtained, we calculate the throughput and bandwidth in Alg. 3 lines 30 and 31. Lastly, the storage size will also be updated into the $VMCapMat$ (i.e., Alg. 3 line 32).

Furthermore, since it gets harder to evaluate demands of different recourse types together (because they have different units), AutoTiering normalizes the VM's estimated/measured throughput, bandwidth and storage utilization value according to the total available resource capacity of each tier, which is called the normalized capacity utilization rate matrix $VMCapRatMat$, as shown in Alg. 4 lines 4-13.

#### 4.1.3.3 Performance Score Calculation

AutoTiering takes three steps to calculate the performance score based to reflect the following factors: First, *Characteristics of both tier and VM*: the score should reflect each tier's specialty and each VM's workload characteristics running on each tier. Thus, our solution is to calculate each VM's score on each tier separately. Second, *SLA weights*: VMs are not equal since they have different SLA weights, as shown in Eq. 4.1. Third, *Confidence of estimation*: we use coefficient variation calculated in performance matrices to reflect the confidence of estimation. Fourth, *History and migration costs*: a comprehensive aging factor for historical scores and estimated migration cost are considered during the score calculation.

*[Step 1] Tier Specialty Matrix*: To reflect the specialty, we introduce a two-dimension tier-specialty matrix $spc$. For example, "$spc[t].P = 1$, $spc[t].B = 1$ and $spc[t].S = 0$" reflects that tier $t$ is good at throughput and bandwidth, but bad at storage capacity. In fact, this matrix can be extended to a finer granularity to further control specialty degree, and more types of resources can be included into this matrix, if needed. Moreover, tiers are sorted in the order of high-to-low-end (e.g., most-to-least-expensive-tier) in the matrix, and this order is regarded as a priority order during the migration decision making period.

*[Step 2] Orthogonal Match between VM Demands and Tier Specialties:* The next question is "*how to reflect each VM's performance on each tier AND reflect how good VMs can utilize each tier's specialty?*". Our solution is to introduce a process called "orthogonal match" (denoted as "Ω") to score the "matching degree". This process is a per-VM-per-tier multiplication operation of

"specialty" matrix and "$VMCapRatMat$" matrix, i.e.,

$$\begin{aligned}
&currCapScore(t,v) = \Omega(t,v) \\
&= \Big[spc[t].P \times wetP[t], \quad spc[t].B \times wetB[t], \quad spc[t].S \times wetS[t]\Big] \\
&\times \begin{bmatrix} VMCapRatMat[t][v].P \\ VMCapRatMat[t][v].B \\ VMCapRatMat[t][v].S \end{bmatrix} \times VM[v].SLA \times VM[v].conf \\
&\div (wetP[t] + wetB[t] + wetS[t]),
\end{aligned} \tag{4.6}$$

where $currCapScore$ gives the current capacity score, and $VMCapRatMat$ is the VM capacity utilization rate matrix.

*[Step 3] Comprehensive Performance Score:* The final performance score is a comprehensive sum of historical score, current epoch capacity score and penalty of corresponding migration cost, i.e.,

$$\begin{aligned}
&tierVMPerfScore = agingFactor \times histTierVMPerfScore \\
&+ currCapScore - wetMig \times migCost
\end{aligned} \tag{4.7}$$

This process is also shown in Alg. 4 line 14. Specifically, to avoid the case that some VMs are frequently migrated back and forth between tiers (due to making decision only based on recent one epoch which may contain I/O spikes or bursties), AutoTiering needs to comprehensively consider history scores, with a preset $agingFactor$ to fadeout outdated scores. Current capacity score $currCapScore$ is calculated by the orthogonal match procedure. Additionally, Alg. 4 lines 16-18 show the procedure of $migrationCost$ calculation.

### 4.1.4 Evaluation

#### 4.1.4.1 Evaluation Methodology

**4.1.4.1.1 Implementation Details** We build AutoTiering on VMware ESXi hypervisor 6.0.0 [17]. Table 4.2 summarizes the server configuration of our implementation. Table 4.3 further shows the specs of each tier (each tier has multiple SSDs). We set the specialty matrix such that tier 1 is good for throughput and bandwidth performance, tier 2 is the secondary performance tier but with larger capacity, and tier 3 is the capacity tier to replace HDDs.

**4.1.4.1.2 Workloads** To evaluate performance under different algorithms, we use IOMeter [151] and FIO [152] to generate I/O workloads to represent real world use cases. Table 4.4 shows some statistical analysis of 14 used workloads. Each VM has multiple VMDKs with different sizes, such as system disk, datastore disk, etc.

**4.1.4.1.3 Comparison Candidates** We compare AutoTiering (*AT*) with two other solutions [125]: (1) *IDT*: IOPS Dynamic Tiering, implements dynamic configuration and placement using a greedy IOPS-only criteria where the higher IOPS extents are moved to the higher IOPS tiers; and (2) *EDT*: Extent-based Dynamic Tiering, updates VM-tier assignment for every epoch, based on each VM's capacity and IOPS requirements. To fully utilize the high speed all-flash datacenter, we slightly modify IDT and EDT to support the "*per-VMDK-based*" operation.

| **Component** | **Specs** |
|---|---|
| Host Server | HPE ProLiant DL380 G9 |
| Host Processor | Intel Xeon CPU E5-2360 v3 |
| Host Processor Speed | 2.40GHz |
| Host Processor Cores | 12 Cores |
| Host Memory Capacity | 64GB DIMM DDR4 |
| Host Memory Data Rate | 2133 MHz |
| Host Hypervisor | VMware ESXi 6.0.0 |

Table 4.2: Host server configuration.

| **Tier** | **Model** | **Protcl.** | **IOPS** | | **MBPS** | | **PerDisk** |
|---|---|---|---|---|---|---|---|
| | | | **R** | **W** | **R** | **W** | **Size(GB)** |
| 1 | Samsung PM953 | NVMe | 240K | 19K | 1000 | 870 | 480 |
| 2 | Samsung PM1633 | SAS | 200K | 37K | 1400 | 930 | 960 |
| 3 | Samsung PM863 | SATA | 99K | 18K | 540 | 480 | 960 |

Table 4.3: Multi-tier flash drivers configuration.

#### 4.1.4.2 Study on Throughput, Bandwidth and Latency Changes

Fig. 4.4 illustrates the average throughput, bandwidth, and normalized latency of all tiers over time for both read (*Rd*) and write (*Wt*) I/Os. AutoTiering achieves up to 44.74% and 38.78% higher IOPS than IDT and EDT, respectively. Similar results can be obtained for bandwidth and latency as shown in Figs. 4.4(b) and (c). Fig. 4.5 depicts per-tier results to further show the performance improvement brought by AutoTiering.

| **Load** | **Workload** | **Represented Scenarios** | **Thrupt. (IOPS)** | **BW. (BPS)** |
|---|---|---|---|---|
| Heavy | BasicVerify | SQL database server | 95.5K | 373M |
| | SSDSteady | System development | 116K | 453M |
| | Zipf IOs | Web apps | 1942K | 7585M |
| | AsyncRead | Read intensive apps | 88.3K | 345M |
| | AsyncWrite | Write intensive apps | 6.65K | 25M |
| Middle | Flow | Big data frameworks | 19.2K | 150M |
| | IOmeter | File server | 47K | 205M |
| | JESD | High endurance apps | 18.3K | 136M |
| | LatencyProfile | Cloud system manager | 39.6K | 155M |
| | SSDTest | Hardware development | 47K | 205M |
| Light | RandZone | Multi-user database | 7.75K | 30.3M |
| | SurfaceScan | Enterprise backup server | 6.98K | 436M |
| | SyncRead | Read intensive sync apps | 6.65K | 25M |
| | SyncWrite | Metadata sync server | 4 | 16K |

Table 4.4: Resource demands of selected workloads.

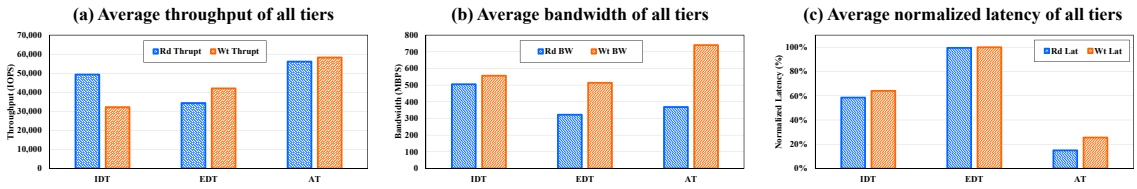



Figure 4.4: Average throughput, bandwidth, and latency of all tiers.

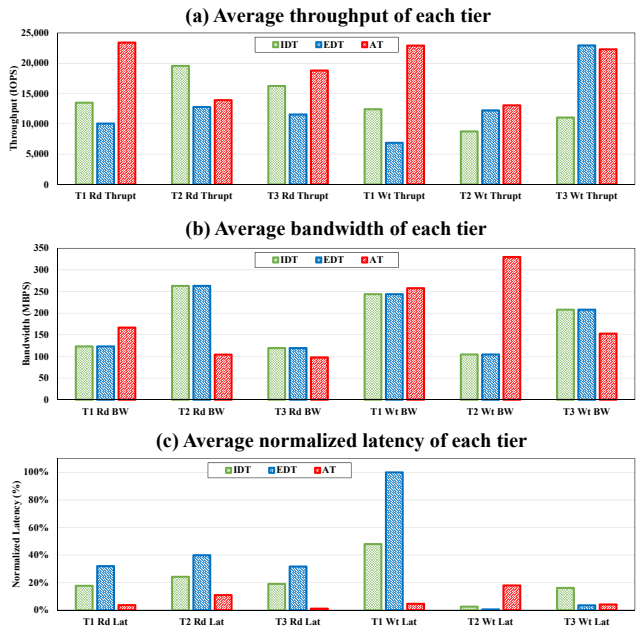



Figure 4.5: Average throughput, bandwidth, and latency of each tier.

We observe that AutoTiering performs the best in terms of (both read and write) throughputs, bandwidths and latencies on tier 1, which is because the specialty matrix sets tier 1 to optimize performance-sensitive workloads. On the other hand, we also see that AutoTiering sometimes achieves lower throughput and bandwidth in tier 2 and 3 compared with IDT and EDT. This is because IDT is an IOPS-only algorithm, which migrates high-IOPS-demand (especially write I/O) workloads to tier 1, such that the write IOPS is optimized. Similarly, EDT considers both IOPS and capacity, and thus has slightly better write IOPS compared to AutoTiering in the capacity tier 3. It is worth mentioning that AutoTiering achieves the lowest latencies in all cases except write latency in tier 2 (as shown in Fig. 4.5(c) 5th column), because AutoTiering migrates many VMDKs that have large average I/O size (high write bandwidth), and thus, as a tradeoff, the latency is increased.

Moreover, Fig. 4.6 depicts the distribution of total throughput and bandwidth of all tiers for different algorithms. From Fig. 4.6(a), we observe that under AutoTiering (red curve), majority of I/Os has more than 100K IOPS, and even half of them have more than 125K IOPS. In contrast, 90% of I/Os are less than 100K IOPS (blue curve) under IDT, and almost all I/Os from IDT are less than 100K (green curve). Similarly, from Fig. 4.6(b), we can see that the majority (around 90%) of IDT and EDT I/Os are less than 1,200 MBPS, while more than half of AutoTiering's I/Os can achieve larger than 1,200 MBPS bandwidth.

#### 4.1.4.3 Study on Runtime Distribution of Resource Utilization

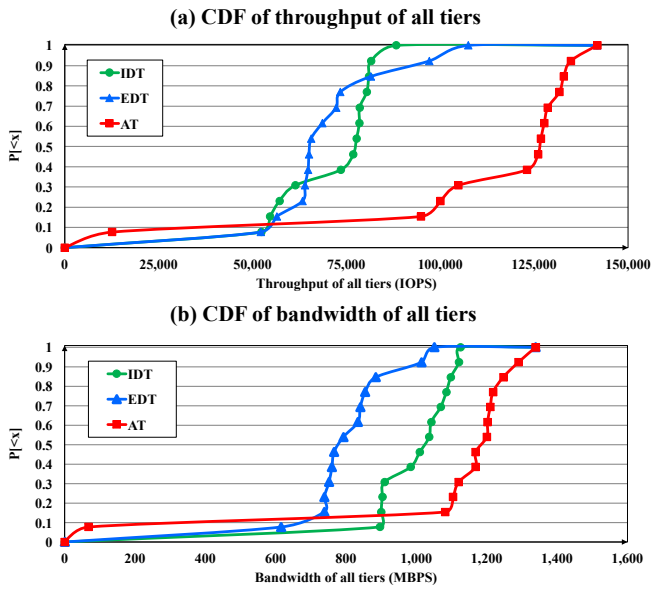



Figure 4.6: CDF of throughput and bandwidth of all tiers.

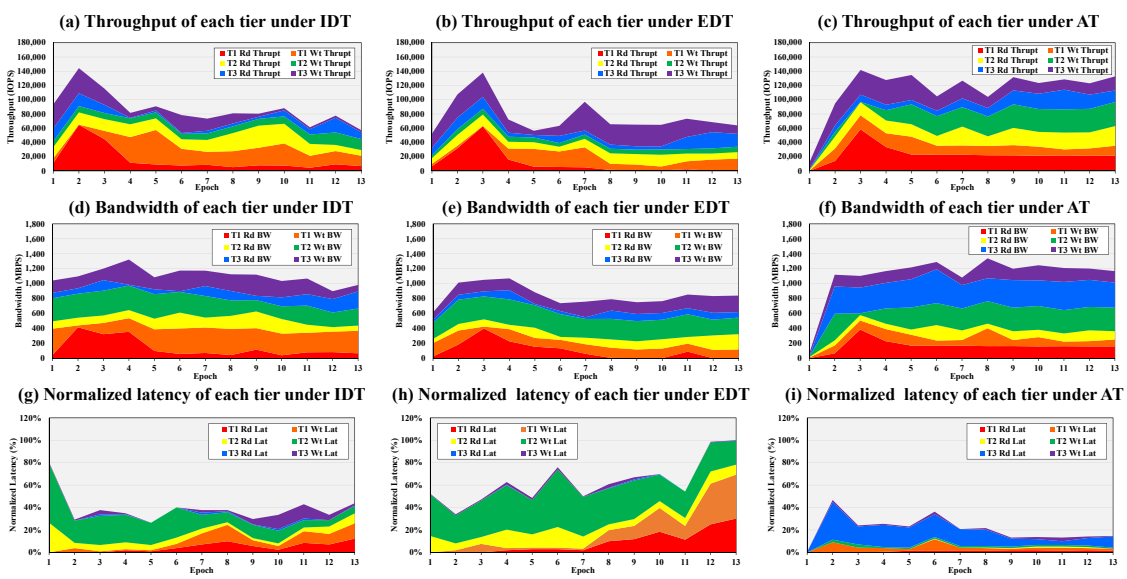



Figure 4.7: Runtime changes of throughput, bandwidth, and latency of each tier under different algorithms.

Fig. 4.7 shows runtime changes of throughput, bandwidth and latency distribution across

tiers over time. We observe that the areas in Figs. 4.7(c) and (f) are larger than those in Figs. 4.7(a)-(b), and Figs. 4.7(d)-(f), respectively. The areas in Fig. 4.7(i) are also much smaller than those in Figs. 4.7(g)-(h). This verifies our observations in Sec. 4.1.4.2 that AutoTiering achieves better throughput and bandwidth performance than IDT and EDT. We also observe in Figs. 4.7(a) to (f) that the area of each tier in AutoTiering is "thicker" than that in IDT and EDT (after AutoTiering's warming up periods from the 0-$th$ to the 3-$th$ epoch). This indicates that AutoTiering can better utilize throughput and bandwidth resources of each tier by proper and less migrations. In Fig. 4.7(i), we see that the majority of AutoTiering's latency is located in tier 3 (the write latency "*T3 Rd Lat*"), which is because that tier 3 is regarded as the capacity tier to replace HDD. As a result, AutoTiering migrates read-intensive VMs with large VMDKs to leverage tier 3, and leaves tiers 1 and 2 for other write-intensive workloads.

#### 4.1.4.4 Study on Migration Overhead

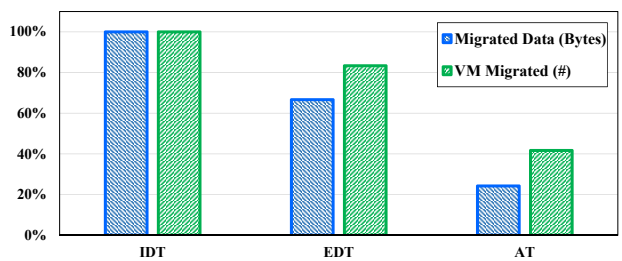



Figure 4.8: Normalized migration cost results.

To investigate the migration overhead of three algorithms, we show the normalized temporal migration cost results in Fig. 4.8. The blue bars show the normalized total migrated data size, and the green bars show the normalized number of VMs that are migrated (multiple migrations on a single VM is counted as 1). The former is to reflect the "working volume size", and the latter is to reflect the "working set size". AutoTiering performs best among the three, since it migrates less data and interrupts less VMs, which saves lots of system resources. In summary, AutoTiering chooses the best tier for each VM for better performance and prevents unnecessary migrations due to I/O spikes, ascribed to its comprehensive decision which is based on a more accurate performance estimation method.

### 4.1.5 Summary

We present a novel data placement manager "AutoTiering" to optimize the virtual machine performance by allocating and migrating them across multiple SSD tiers in the all-flash datacenter. AutoTiering is based on an optimization framework to provide the global best migration and allocation solution over runtime. We further proposed an approximation algorithm to solve the problem in a polynomial time, which considers both historical and predicted performance factors, and estimated migrating cost. Experimental results show that AutoTiering can significantly improve system performance.

## 4.2 Endurance, Total Cost of Ownership and Workload Deployment

### 4.2.1 Motivation

The world has entered the era of "Big Data", when large amount of data is being collected from a variety of sources, including computing devices of all types, shapes and forms. This data is then being pushed back to large, back-end datacenters where it is processed to extract relevant information. As a result of this transformation, a large number of server-side applications are becoming increasingly I/O intensive. Furthermore, with the amount of data being gathered increasing with every passing day, the pressure on the I/O subsystem will continue to keep on increasing [153].

To handle this high I/O traffic, datacenter servers are being equipped with the best possible hardware available encompassing compute, memory, networking and storage domains. Traditionally, I/O has been handled by HDDs. HDDs have the benefit of providing an excellent economic value ($/GB), but being built with mechanical moving parts, they suffer from inherent physical throughput limitations, especially for random I/Os. To counter these performance limitations, SSDs have recently begun to emerge as a viable storage alternative to HDDs. In the recent past, SSDs have gained widespread adoption owing to reduced costs from the economies of scale. Datacenters, especially popular public cloud providers (e.g., [154, 155]) have been at the forefront of adopting flash technology.

Nevertheless, during this revolutionary change in cloud storage systems, flash based SSDs face two major concerns: cost and write amplification (WA). Firstly, the costs of owning (purchasing and maintaining) SSDs can still be very high. Balancing the trade-off between performance and economy is still an uphill battle. Currently, Total Cost of Ownership (TCO), comprising of two major costs, i.e., Capital and Operating Expenditures, remains a popular metric. However, only a few prior

studies have focused on the TCO of SSDs in datacenters, especially with the consideration of the cost of SSD's endurance.

Secondly, SSDs have limited write cycles and also suffer from Write Amplification (WA) which is caused by a number of factors specific to flash devices including erase-before-rewrite, background garbage collection, and wear leveling. In fact, the WA of an SSD is a direct function of the I/O traffic it experiences. The I/O traffic, in turn, comprises of a number of different factors like the fraction of writes (as opposed to reads), the average size of I/O requests, the arrival rate of I/Os, and the ratio of sequential I/O patterns (as opposed to random I/O) in the overall I/O stream. Greater WA can significantly reduce the lifetime and increase the ownership cost of flash devices.

Therefore, in this work, we focus on the problem of deploying and allocating applications to a shared all-flash storage system of modern datacenters in order to reduce WA and TCO. In detail, workloads (I/O streams from an application) have different features, but in a long term of view, the I/O pattern of the same application can be characterized. Hence, we can address the above two concerns by investigating the relationship between workload patterns and WA and then leveraging the relationship to develop new TCO models. In our experiments, we find that workloads with different sequential ratios have varying write amplifications even on the same SSD, which changes the lifetime of the device and eventually affects the TCO. We are thus motivated to evaluate storage systems from a cost perspective that includes many dimensions such as maintenance and purchase cost, device wornout, workload characteristics, and total data amount that can be written to the disk, etc. To sum up, in this work, we make the following contributions to achieve this goal.

- We conduct real experiments to measure and characterize the write amplification under different workloads, and reveal the relationship between write amplification and workload sequential ratio for each disk with fixed flash Translation Layer (FTL) specs.

- We propose a new TCO model to consider multiple factors like SSD lifetime, workload sequentiality, write wornout and the total writes.

- We propose statistical approaches for calculating components that are essential for computing the TCO but cannot (practically) be measured from SSDs during runtime, such as write amplification and wornout of each SSD.

- Based on our TCO model, we develop a set of new online adaptive flash allocation managers called "MINTCO", which leverage our TCO model to dynamically assign workloads to the SSD disk

pool. The goals of MINTCO are: (1) to minimize the TCO, (2) to maximize client throughput as many as possible, and (3) to balance the load among SSD devices and best utilize SSD resources.

Lastly, we evaluate our new models and approaches using real world trace-driven simulation. Our experimental results show that MINTCO can reduce the TCO by up to 90.47% compared to other traditional algorithms. Meanwhile, it guarantees relatively high throughput and spatial utilization of the entire SSD-based datacenter.

### 4.2.2 TCO Models of All-Flash Storages

TCO of an all-flash datacenter's storage system is a mix of a large number of items. Broadly speaking, these items can be broken down into two major categories: (1) Capital Expenditure (CapEx), and (2) Operating Expenditure (OpEx). Capital Expenditure refers to the amount of money that needs to be spent in setting up a facility. These include the cost of buying individual components of the servers, power supplies, racks that house the servers, among other things. OpEx, on the other hand, is the amount of money that is spent in the day-to-day operation of a datacenter. Examples of OpEx include electricity costs and personnel costs (required for maintaining the datacenter). CapEx is traditionally a large, one time expenditure [156] while OpEx consists of small(er), recurring expenditures.

In this subsection, we develop a TCO model for an SSD intensive datacenter, based on the characteristics of the workloads (i.e., application I/O streams) that are scheduled on to those SSD devices. Our TCO model focuses specifically on the costs related to acquiring (i.e., CapEx) and maintaining (i.e., OpEx) SSDs in a datacenter.

#### 4.2.2.1 Workload and Storage Models

First, we briefly explain our assumptions about datacenters, their workloads and storage systems. We assume the datacenter to be a large pool of SSD devices. This helps us abstract the problem of modeling SSDs from a per-server resource to a pool of datacenter-wide resources. We then model the storage system of such a datacenter as a *workload-to-disk allocation problem*, as shown in Fig. 4.9. In this model, we have a pool of $N_D$ SSDs as shown in Fig. 4.9. Meanwhile, there are $N_W$ applications (workloads) that submit I/O requests with logical write rates $\lambda_{L_{J_i}}$ (where $1 \leq i \leq N_W$, and "$L_J$" stands for "logical" and "job"), as seen in the left hand box of Fig. 4.9. To complete the connection between I/O requests and the SSDs, we assume an allocation algorithm that is used by the dispatcher to assign I/O workloads to different SSDs. Each workload has its own

characteristic, and arrives at the dispatcher at different times. Multiple workloads can be assigned to the same disk as long as the capacity (e.g., space and throughput) of the disk is sufficient, such that the logical write rate ($\lambda_{L_i}$) to disk $i$ is the summation of logical write rates from the workloads in the set $\mathbb{J}_i$ that are allocated to that SSD, i.e., $\lambda_{L_i} = \sum_{j \in \mathbb{J}_i} \lambda_{L_{J_j}}$. We summarize our main assumptions as follows.

**4.2.2.1.1 I/O Workload with Certain Properties** "Workload" is defined as an endless logical I/O stream issued by applications. Particularly, from a long-term view (e.g., years), characteristics of workloads, such as sequential ratio, daily write rate, read-write ratio, working set size, re-access ratio, can be abstracted as (almost) fixed values. Workloads may arrive at the datacenter at different times. Once a workload arrives, the dispatcher assigns it to one certain disk, and the disk will execute this workload until the disk(s) is (are) "dead" (i.e., SSD write cycle limit is reached), or the workload finishes. We ignore the overhead (such as time and energy consumption) during the workload deployment.

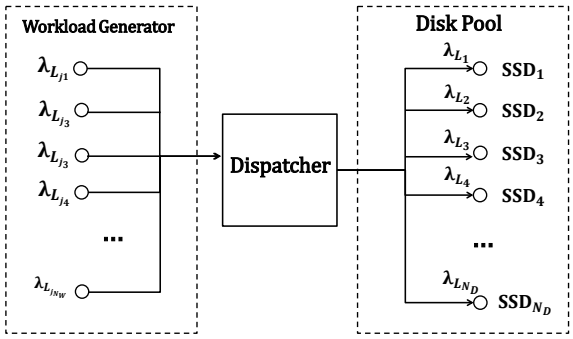



Figure 4.9: Model of a datacenter storage system.

**4.2.2.1.2 Isolation among Multiple Workloads on a Single SSD** Multiple workloads can be assigned to a single SSD, and have separate and independent working sets (i.e., address spaces and segments are isolated). Therefore, the cross-workloads effects along I/O path due to interleaving working sets are negligible.

**4.2.2.1.3 SSD's Write Amplification Model** We use the WA model to capture the behavior of an SSD under a workload with a specific I/O pattern. Our WA model can estimate the WAF of each disk by using the sequentiality information of multiple workloads that are concurrently executing at

a particular SSD. The write wornout of each SSD can further be estimated using the runtime status of that SSD's WA.

#### 4.2.2.2 Total Cost of Ownership Model

Owning and maintaining a low-cost SSD-intensive datacenter is critically important. TCO has been widely adopted to evaluate and assess storage subsystem solutions for traditional hard drives. However, to the best of our knowledge, there is no standard formula for calculating the TCO of the SSD-intensive storage subsystem. In order to comprehensively assess the expenditure of a datacenter, a generic TCO model should consider purchasing and maintenance costs, service time, served I/O amount and device wornout. We present the following models to calculate the TCO. As we mentioned, two major types of costs: CapEx ($C_{I_i}$) and OpEx ($C_{M_i}$) are considered in the basic TCO model. In detail, $C_{I_i} = C_{Purchase_i} + C_{Setup_i}$ and $C'_{M_i} = C'_{Power_i} + C'_{Labor_i}$, where $C_{Purchase_i}$ and $C_{Setup_i}$ are one-time cost (\$) of device purchase and device setup, and $C'_{Power_i}$ and $C'_{Labor_i}$ are power and maintenance labor cost rate (\$/day). Although CapEx ($C'_{I_i}$) is one time cost, OpEx ($C'_M$) is a daily rate and the TCO depends on the amount of time that an SSD has been used. Therefore, we need to attach a notion of *time* to TCO. Assume we know the expected life time ($T_{Lf_i}$) of each disk (i.e., $T_{Lf_i} = T_{D_i} - T_{I_i}$, where $T_{D_i}$ and $T_{I_i}$ are the time when the disk $i$ is completely worn out and the time when it starts to serve its first request, respectively), the total cost for purchasing and maintaining a pool of SSDs can be calculated as:

$$TCO = \sum_{i=1}^{N_D} (C_{I_i} + C'_{M_i} \cdot T_{Lf_i}), \tag{4.8}$$

where $N_D$ is the number of disks in the pool. Fig. 4.10(a) also illustrates an example from time stamp view, where I/O workloads keep arriving and thus the physical write rate of disk $i$ increases accordingly. Furthermore, to conduct a fair judgment, we introduce the data-averaged TCO rate ($TCO'$) from the perspective of the cost vs. the total amount of (logical) data served (written) to an SSD as follows.

Therefore, to conduct a fair and meaningful comparison, we introduce the data-averaged TCO rate ($TCO'$) from the perspective of the cost vs. the total amount of (logical) data served (written) to an SSD as follows.

$$TCO' = \frac{\sum_{i=1}^{N_D} (C_{I_i} + C'_{M_i} \cdot T_{Lf_i})}{\sum_{j=1}^{N_W} D_j}, \tag{4.9}$$

where $\sum_{j=1}^{N_W} D_j$ is the total *logical* data write amount for all $N_W$ workloads. We use logical writes as a proxy for physical writes not only because the former is much easier to obtain for most workloads, but also because by being normalized by the logical writes, the $TCO'$ is able to reflect the WAF and judge the disk-level wear leveling performance of different allocation algorithms.

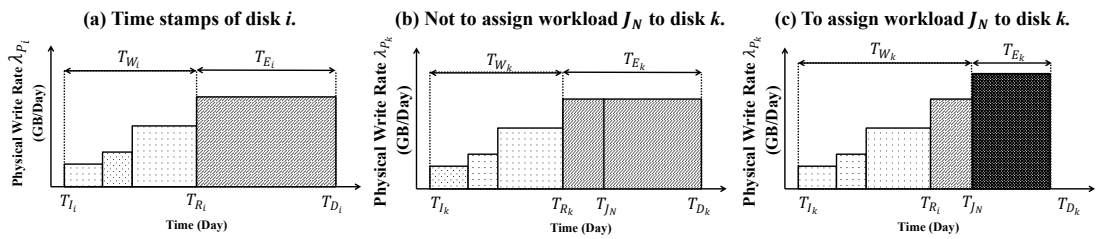



Figure 4.10: Examples of physical write rates vs. workload arrivals.

#### 4.2.2.3 Calibrating TCO Models

The models developed in the prior subsection have all assumed that certain parameters for TCO calculation (e.g., total logical data write amount, expected lifetime, etc.) are readily available or measurable. However, it is impractical to measure some parameters that are necessary for our TCO models. In this subsection, we propose a mathematical approach of estimating those parameters that are hard to be measured directly.

**4.2.2.3.1 Total Logical Data Writes** Given workload $j$'s logical write rate $\lambda_{L_j}$, arrival time $T_{A_j}$ and estimated time of death ($T_{D(j)}$) of workload $j$ 's host disk $D(j)$, we can calculate the total amount of data written by all the jobs over their course of execution as: $\sum_{j=1}^{N_W} D_j = \sum_{j=1}^{N_W} \lambda_{L_j}(T_{D(j)} - T_{A_j})$, where $\lambda_{L_j}$ is the logical data write rate of workload $j$. The only unknown parameter left is $T_{D(j)}$, which can be obtained by calculating each host disk's expected lifetime.

**4.2.2.3.2 Expected Lifetime** The lifetime of a disk depends not only on the write traffic from the currently executing jobs, but also on those jobs that have already been deployed on the disk. Furthermore, we also need to account for the effects of the *combined* write traffic of the workloads that are concurrently executing on a disk. As shown in Fig. 4.10(a), the lifetime of disk $i$ is the period from $T_{I_i}$ to $T_{D_i}$. We further split the lifetime into two phases: (1) all (accumulated) working epochs ($T_{W_i}$) until the last workload arrives, i.e., $T_{W_i} = T_{R_i} - T_{I_i}$, where $T_{R_i}$ is the assigned time of the most recent workload; and (2) expected work lifetime ($T_{E_i}$) from $T_{R_i}$ to the expected death time, i.e.,

$T_{E_i} = T_{D_i} - T_{R_i}$. The former is easy to monitor, and the latter is further estimated as the available remaining write cycles of disk $i$ divided by the physical data write rate ($\lambda_{P_i}$) of disk $i$ from $T_{Ri}$. Moreover, $\lambda_{P_i}$ can be calculated as disk $i$'s logical data write rate ($\lambda_{L_i}$) times disk $i$'s WAF ($A_i$). Thus, we have $T_{Lf_i} = T_{D_i} - T_{I_i} = T_{W_i} + T_{E_i} = (T_{R_i} - T_{I_i}) + \frac{W_i - w_i}{\lambda_{P_i}} = (T_{R_i} - T_{I_i}) + \frac{W_i - w_i}{\lambda_{L_i} \cdot A_i} = (T_{R_i} - T_{I_i}) + \frac{W_i - w_i}{\lambda_{L_i} \cdot f_{seq}(S_i)}$, where $A_i$, $W_i$, $w_i$ and $S_i$ are the WAF function, the total write limit, current write count (wornout), and sequential ratio of all running workloads of disk $i$, respectively. Since the SSDs' hardware are fixed, we denote $A_i$ as a function of workload's write I/O sequential ratio ($f_{seq}$) of disk $i$, which will be validated and regressed in our experimental subsection (Sec. 4.2.4.1.5). In fact, we can plug any WAF model into this TCO model. As of now, we also know $T_{R_i}$, $T_{I_i}$ and $W_i$, and what we need to estimate next are the remaining parameters, i.e., $\lambda_{L_i}$, $S_i$ and $w_i$.

**4.2.2.3.3 Logical Write Rate of Workloads on Disk** For disk $i$, its logical write rate $\lambda_{L_i}$ should be the sum of all its assigned workloads' logical write rates, i.e., $\lambda_{L_i} = \sum_{j \in \mathbb{J}_i} \lambda_{L_{ij}}$, where $\mathbb{J}_i$ is the set of workloads running on disk $i$. Notice that there is a boundary case during the very early stage when no workloads have been assigned to the disk $i$ (i.e., $\mathbb{J}_i = \emptyset$), such that $\frac{W_i - w_i}{\lambda_{L_i} \cdot f_{seq}(S_i)}$ becomes infinite. To avoid such an extreme case, we conduct a warming up process that assigns at least one workload to each disk. Only after this warming up phase is done, we start to calculate $T_{Lf_i}$.

**4.2.2.3.4 Sequential Ratio of Workloads on Disk** In order to calculate the write amplification $A_i$ in Sec. 4.2.2.3.2, we need to know the sequential ratio of multiple workloads that are assigned to one disk. Unlike the logical write rate, the combined sequential ratio of multiple workloads is not equal to the sum of sequential ratios of all workloads. Our estimating solution is to assign a weight to each workload stream's sequential ratio and set the weight equal to the workload's logical data write rate. Hence, for multiple workloads running on the disk, we can calculate the overall sequential ratio as: $S_i = \frac{\sum_{j \in \mathbb{J}_i} \lambda_{L_{ij}} S_{ij}}{\sum_{j \in \mathbb{J}_i} \lambda_{L_{ij}}}$, where $\lambda_{L_{ij}}$ and $S_{ij}$ are the logical write rate and sequential ratio of $j$th workload running on disk $i$.

**4.2.2.3.5 Write Wornout Count of Disk** The last item we need to estimate is the current physical write count $w_i$ (in Sec. 4.2.2.3.2) inside each SSD device. It is hard to exactly measure the overall write count of an SSD during its lifetime. However, we can estimate the current write count by adding the estimated write counts of all the workloads over all past epochs. For each epoch, we multiply the total logical write rate by the corresponding WAF to get the physical write rate. By iterating this process for all epochs, we can finally get the total write wornout count for each disk.

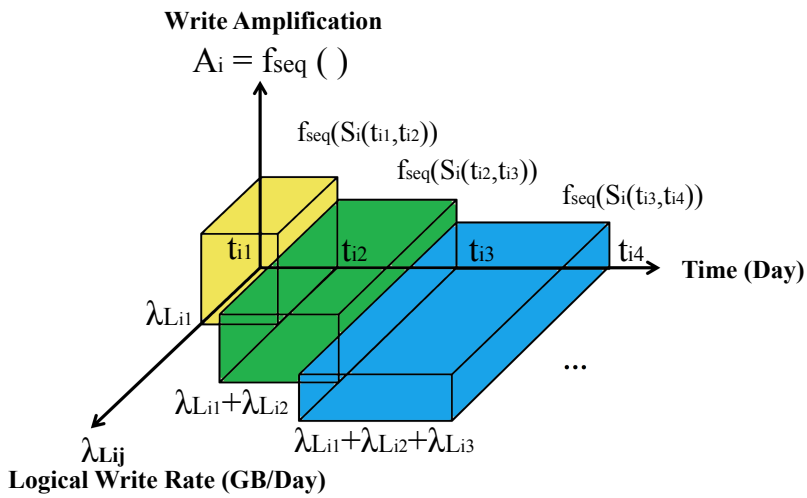



Figure 4.11: An example of write wornout count estimation.

Fig. 4.11 shows a simple example of estimating a disk's write wornout when there are multiple workloads executing on disk $i$. Each brick represents an epoch, which is bounded by its workloads' allocation times. The volume of all these bricks gives the total write wornout count ($w_i$) for disk $i$. To calculate $w_i$, we further convert above-mentioned $\lambda_{L_i}$ and $S_i$ to the total logical data write rate function and the sequential ratio function during each epoch $[t_{ix}, t_{i(x+1)})$, respectively: $\lambda_{L_i}(t_{ix}, t_{i(x+1)}) = \sum_{j \in \mathbb{J}_i(t_{ix}, t_{i(x+1)})} \lambda_{L_{ij}}$, and $S_i(t_{ix}, t_{i(x+1)}) = \frac{\sum_{j \in \mathbb{J}_i(t_{ix}, t_{i(x+1)})} \lambda_{L_{ij}} S_{ij}}{\sum_{j \in \mathbb{J}_i(t_{ix}, t_{i(x+1)})} \lambda_{L_{ij}}}$, where $x$ is the number of workloads executing on disk $i$, and $t_{ix}$ is the arrival time of disk $i$'s $x^{th}$ workload. $\mathbb{J}_i(t_{ix}, t_{i(x+1)})$ is the set of workloads running on disk $i$ during $t_i x$ and $t_{i(x+1)}$ epoch. $\lambda_{L_{ij}}$ and $S_{ij}$ are the write rate and sequential ratio of $j^{th}$ workload in $\mathbb{J}_i(t_{ix}, t_{i(x+1)})$. Therefore, wornout $w_i$ can be calculated as: $w_i = \sum_{t_{ix} \in T_i} [\lambda_{L_i}(t_{ix}, t_{i(x+1)}) \cdot f_{seq}(S_i(t_{ix}, t_{i(x+1)})) \cdot (t_{i(x+1)} - t_{ix})]$. Here, $t_{ix}$ is the starting time of disk $i$'s $x^{th}$ epoch. At the sample moment $t_{i(x+1)}$, we assume there are $x$ workloads running on disk $i$. $\mathbb{T}_i$ is the set of arrival times of each workload running on disk $i$. The three parts ($\lambda$, WAF and time) match the three axes from Fig. 4.11, where each brick stands for each epoch, and the total volume of these bricks is the accumulated write count value of that SSD disk. Therefore, the data-avg TCO rate $TCO'$ in Eq. 4.9 can be calibrated as:

$$TCO' = \frac{\sum_{i=1}^{N_D} [C_{I_i} + C'_{M_i}(T_{W_i} + \frac{W_i - w_i}{\lambda_i \cdot A_i})]}{\sum_{j=1}^{N_W} \lambda_j (T_{Lf_{D(j)}} - T_{I_j})}. \tag{4.10}$$

### 4.2.3 Algorithm Design

Based on the proposed TCO model, we further design a set of online allocation algorithms, called "MINTCO", which adaptively allocate new workloads to SSDs in the disk pool. The main goal is to minimize the data-avg TCO rate ($TCO'$) of the storage pool and also to conduct disk-level wear leveling during workload deployment and allocation.

#### 4.2.3.1 Baseline minTCO

The main functionality of the baseline version of MINTCO is presented in Alg. 5. When a new workload arrives, MINTCO calculates the data-avg TCO rate for the entire disk pool, and then allocates the workload to the SSD that makes the lowest data-avg TCO rate of the entire disk pool.

**Algorithm 5:** minTCO

```
1  Procedure minTCO()
2      for incoming new workload J_N do
3          for i ← 1 to N_D do
4              TCO_List[i] = TCO_Assign(i, J_N);
5          SelectedDisk = TCO_List.minValueIndex();
6          Disk[SelectedDisk].addJob(J_N);
7      return;
8  Procedure TCO_Assign(i,J_N)
9      for k ← 1 to N_D do
10         C_I = getCostInit(k);
11         C_M = getCostMaint(k);
12         if k == i then
13             T_{W_k} = T_{J_N} − T_{I_k};
14             T_{E_k} = getExpFutureWorkTime(k, J_N);
15             Data = getTotalLogWriteAmt(T_{W_k} + T_{E_k}) + (T_{W_k} + T_{E_k} − T_{J_N}) ∗ λ_{J_N};
16         else
17             T_{W_k} = T_{R_k} − T_{I_k};
18             T_{E_k} = getExpFutureWorkTime();
19             Data = getTotalLogWriteAmt(T_{W_k} + T_{E_k});
20         TCO+ = C_I + C_M ∗ (T_{W_k} + T_{E_k});
21         TotalData+ = Data;
22     return TCO/TotalData;
```

Specifically, there are two cases during the calculation of the expected lifetime and the total logical write amount. The first case is that when a new workload is assigned to disk $k$, we use this new workload's arrival time as the boundary between $T_{W_k}$ and $T_{E_k}$ phases, as shown in Alg. 5 lines 13 to 15 and Fig. 4.10(c). The second case is that when the new workload is not assigned to disk $k$ (Alg. 5 lines 17 to 19), we use $T_{R_k}$ (the arrival time of the most recent workload on disk $k$) as the boundary between $T_{W_k}$ and $T_{E_k}$ phases, as shown in Fig. 4.10(b).

As discussed previously, our TCO model is compatible with any WAF models. Here we adopt the WAF model described in Eq. 4.10 to implement the functions in Alg. 5 line 14, 15, 18 and 19. The baseline MINTCO also needs to consider other resource constraints. For example, MINTCO needs to further check if the available spatial (in GB) and throughput (in IOPS) capacities of the chosen SSD are large enough to hold the new workload's working set. If not, MINTCO moves to the next SSD which has the second lowest data-avg TCO rate. If all disks do not have enough capacity, the workload will be rejected.

#### 4.2.3.2 Performance Enhanced minTCO

One limitation of the baseline MINTCO is that it does not balance the load across the SSDs in the pool and thus cannot achieve optimal resources utilization. However, best using of resources (e.g., I/O throughput and disk capacity) is an important goal in real storage system management. To address this limitation, we further develop the performance enhanced manager, namely MINTCO-PERF, which considers statistical metrics (i.e., load balancing and resource utilization) as the performance factors in workload allocation.

**4.2.3.2.1 System Resource Utilization** We consider two types of resources, throughput (IOPS) and space capacity (GB), and calculate the utilization ($U(i,k)$) of disk $i$ when disk $k$ is selected to serve the new workload $J_N$, as:

$$U(i,k) = \begin{cases} \frac{R_U(i)}{R(i)}, & i \neq k \\ \frac{R_U(i)+R(J_N)}{R_i}, & i = k \end{cases}, \tag{4.11}$$

where $R_U(i)$, $R(i)$ and $R(J_N)$ represent the amount of *used* resource on disk $k$, the *total* amount of resource of disk $i$, and the resource *requirement* of workload $J_N$, respectively. When $i$ is equal to $k$, we have the new requirement (i.e., $R(J_N)$) as extra resources needed on that disk. This equation can be used to calculate either the throughput utilization (i.e., $U_p(i,k)$) or the space capacity utilization

(i.e. $U_s(i,k)$). The average utilization can be calculated to represent the system utilization of the entire disk pool: $\overline{U(k)} = \frac{\sum_{k=1}^{N_D} U(i,k)}{N_D}$. Our goal is to increase either the average throughput utilization (i.e., $U_p(i,k)$) or the average space utilization (i.e. $U_s(i,k)$).

**4.2.3.2.2 Load Balancing** We use coefficient of variation ($CV$) of throughput (or space) utilizations among all disks to ensure the load balancing. Specifically, when assigning the workload $J_N$ to disk $k$, we calculate expected $CV(k)$ as: $CV(k) = \frac{\sqrt{\frac{\sum_{i=1}^{N_D} [U(i,k)-\overline{U(k)}]^2}{N_D}}}{\overline{U(k)}}$. A smaller $CV(k)$ indicates better load balancing in the datacenter.

Then, we have our MINTCO-PERF algorithm which aims to minimize the data-avg TCO rate, while achieving best resource utilization and load balancing among disks. MINTCO-PERF uses an optimization framework to minimize the objective function under constrains listed in Eq. 4.12.

$$
\begin{aligned}
&\text{Minimize:}\\
&\quad f(R_w)\cdot TCO'(k)\\
&\quad - g_s(R_r)\cdot \overline{U_s(k)} + h_s(R_r)\cdot CV_s(k)\\
&\quad - g_p(R_r)\cdot \overline{U_p(k)} + h_p(R_r)\cdot CV_p(k)\\
&\text{Subject to:}\\
&\quad i\in D, k\in D\\
&\quad 0\le TCO'(i,k)\le Th_c\\
&\quad 0\le U_s(i,k)\le Th_s\\
&\quad 0\le U_p(i,k)\le Th_p
\end{aligned}
\tag{4.12}
$$

Upon the arrival of a new workload $J_N$, we calculate the "enhanced cost" of the disk pool. The object function in Eq. 4.12 contains the TCO rate cost ($f(R_w)\cdot TCO'(k)$), the resource utilization reward ($g_s(R_r)\cdot\overline{U_s(k)}$ and $g_p(R_r)\cdot\overline{U_p(k)}$), and the load unbalancing penalty ($h_s(R_r)\cdot CV_s(k)$ and $h_p(R_r)\cdot CV_p(k)$). Notice that $TCO'(k)$ and $TCO'(i,k)$ represent the TCO rate of the entire disk pool and the TCO rate of disk $i$, respectively, when disk $k$ is selected to take the workload. Non-negative parameters in Eq 4.12 (e.g., $f(R_w)$, $g_s(R_r)$, $g_p(R_r)$, $h_s(R_r)$ and $h_p(R_r)$) are the weight functions that are related with the read ratio ($R_r = \frac{readIO\#}{totalIO\#}$) and write ratio ($R_w = \frac{writeIO\#}{totalIO\#}$) of workloads. Finally, the disk with the lowest enhanced cost will be selected for the new workload. The reason behind this is that in the real world, write intensive workloads affect WAF and TCO, and

read intensive workloads are sensitive to load balancing degree. In addition, $Th_c$, $Th_s$ and $Th_p$ are used as the upper bounds for TCO, space and throughput resources utilization ratios, respectively.

### 4.2.4 Evaluation

#### 4.2.4.1 Write Amplification Measurement & Modeling

**4.2.4.1.1 Hardware Testbed** Most SSD vendors do not provide APIs or performance counters to measure this physical write quantity. Hence, many prior studies (e.g.,[157, 43]) have tried to develop models for *estimating* the WAF of an SSD based on a certain criterion. We hereby propose to leverage the data *directly measured* from SSDs to calculate a WAF function for an SSD. Our goal is to characterize the effects of write traffic from multiple workloads on the WAF, and see if such a characterization can be generalized as a mathematical model. Table 4.5 and Fig. 4.12(a) show the testbed specification and the WAF measurement workflow, respectively.

| Component | Specs |
|---|---|
| Processor | Xeon E5-2690, 2.9GHz |
| Processor Cores | Dual Socket-8 Cores |
| Memory Capacity | 64 GB ECC DDR3 R-DIMMs |
| Memory Bandwidth | 102.4$GB/s$ |
| RAID Controller | LSI SAS 2008 |
| Network | 10 Gigabit Ethernet NIC |
| Operating system | Ubuntu 12.04.5 |
| Linux Kernel | 3.14 Mainline |
| FIO Version | 2.1.10 with Direct I/O |
| Storage Type | NVMe SSD (Released in 2015) |
| Storage Capacity | 1.6 TB |

Table 4.5: Server node configuration.

**4.2.4.1.2 Filesystem** We test two representative scenarios: “formatting with no file system” and “formatting with Ext4 file system”. (1) “No file system” mimics the use case like a swap partition, where avoiding a filesystem mainly has three advantages: making more of the disk usable, since a filesystem always has some bookkeeping overhead; making the disks more easily compatible with

other systems (e.g., the *tar* file format is the same on all systems, while filesystems are different on most systems.); and enabling enterprise user to deploy customized block manager running on the hypervisor without mounting a traditional filesystem. (2) "Ext4 file system" is a very solid file system which has been widely used for SSDs in datacenters using Linux distributions for the past few years. The journaling that comes with Ext4 is a very important feature for system crash recovery although it causes some acceptable write activity.

**4.2.4.1.3 Precondition** In order to ensure the SSD in the same state and stimulate the drive to the same performance state at the beginning of each measurement, we also conduct a 9 hour "preconditioning process". Table 4.6 shows the detail of our preconditioning setups.

| | **Precon. Seq Fill** | **Precon. Rand Fill** |
|---|---|---|
| RW | Write | Write |
| IODepth | 16 | 32 |
| BlockSize | 1MB | 4KB |
| Job Number | 1 | 4 |

Table 4.6: I/O setups of preconditioning.

In detail, we have the following operations: "Sequential precondition" is that, between each measurement, the SSD is completely fulfilled with sequentially I/Os so that all write I/Os in the measurement workloads are overwrite operations, and WAF results will not be independent on garbage collection. "Random precondition" will further conduct an additional complete overwrite to the device with random I/Os with 4KB granularity after the sequential preconditioning process to randomize the workset distribution. "Rnd-Rnd/Seq-Seq precondition" is the policy that we use the random and sequential precondition for non-100% sequential and 100% sequential I/O workloads, respectively. We attempt to use these workloads to observe the ideal write performance (i.e., steady write performance). These two precondition operations can help us simulate different scenarios.

**4.2.4.1.4 I/O Workloads** In order to study the effects of sequential traffic on WAF, we conduct an experiment that can control the amount of sequential traffic being sent to an SSD. Most workloads in real systems are a certain mixture of sequential and random I/Os. To mimic such real situations, we generate mixed workloads by using an I/O testing tool FIO [152]. We also make changes to a 1.6TB NVMe SSD firmware to parse the value of (page) program and (block) erase counters. We

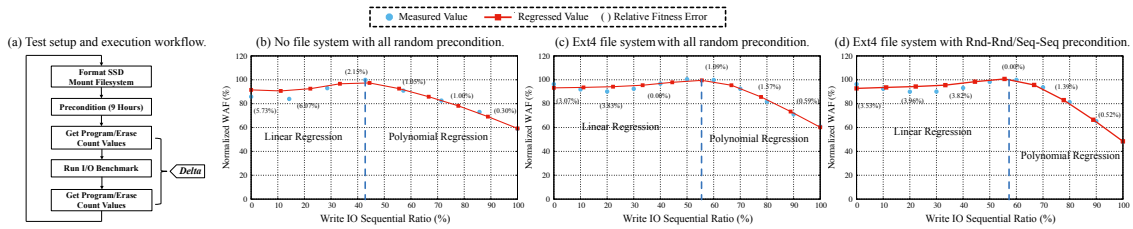



Figure 4.12: WAF measurement experiment results.

investigate the write amplification factor as the ratio of sequential and random accesses, and the changes of these counters, as shown as "delta" in Fig. 4.12(a).

**4.2.4.1.5 WAF Results and Modeling** Figs. 4.12(b)-(d) show three representative cases from our WAF experimental results, which present the normalized WAF as a function of different sequential ratios on write I/Os. The WAF data points are normalized by the largest WAF across different workload sequential ratios (e.g., the WAF under 40.22% sequential ratio in Fig. 4.12 (b)). Thus, the original WAF is $\in [1, +\infty)$, while the normalized WAF is $\in [0, 1]$.

First, we can see that WAF curves in the three figures are similar, i.e., the curves can be regressed into two stages: a flat linear regression stage and a dramatically decreasing polynomial regression stage. The former part shows that the write amplification factor of mixed workloads is almost identical to that of a pure random workload and keeps almost constant before a turning point (around 40% to 60%). But, after this turning point, the WAF dramatically decreases. In another word, a small fraction of random accesses is necessary to intensify the write amplification factor.

We further regress the WAF ($A$) as a piecewise function of sequentiality of I/O operations in the workload as shown in Eq. 4.13, where $\alpha$, $\beta$, $\gamma$, $\mu$ and $\varepsilon$ are parameters, and $S$ is the sequential ratio.

$$A = f_{seq}(S) = \begin{cases} \alpha S + \beta, & S \in [0, \varepsilon] \\ \eta S^2 + \mu S + \gamma, & S \in (\varepsilon, 1] \end{cases} \tag{4.13}$$

At the turning point $S = \varepsilon$, we have $\alpha\varepsilon + \beta = \eta\varepsilon^2 + \mu\varepsilon + \gamma$. Additionally, $\alpha$ is close to zero since the linear regression stage is relatively smooth. We carry out these experiments multiple times on a number of NVMe SSDs, and draw our conclusions as follows. We believe that such a mathematical model of sequential write traffic vs. WAF can be constructed for most devices, and each SSD can have its own unique version of WAF function, depending on a number of its

hardware-related factors (FTL, wear leveling, over-provisioning, etc.). However, being able to regress a mathematical model for the problem forms the basis of the rest of this work. Additionally, we also observe that the regression turning point of the non-filesystem case (Fig. 4.12(b)) comes earlier than Ext4's (Figs. 4.12(c) and (d)). This validates the fact that Ext4's (bookkeeping) overhead is heavier than the raw disk. Moreover, when the sequential ratio is 100%, the WAF under "Rnd-Rnd/Seq-Seq precondition" case (Fig. 4.12(d)) is lower than that under the "All-Rnd precondition" case (Fig. 4.12(c)). This validates that in the former case, the steady write performance can be reached.

#### 4.2.4.2 TCO Evaluation

**4.2.4.2.1 Benchmarks and Metrics** In this subsection, we plug the WAF model regressed from the real experiments to our TCO model, and then evaluate our MINTCO algorithms.

| Trace Name | S (%) | $\lambda$ (GB/day) | $P_{Pk}$ (IOPS) | $R_W$ (%) | WSs (GB) |
|---|---|---|---|---|---|
| mds0 | 31.52 | 21.04 | 207.02 | 88.11 | 6.43 |
| prn0 | 39.13 | 131.33 | 254.55 | 89.21 | 32.74 |
| proj3 | 72.06 | 7.50 | 345.52 | 5.18 | 14.35 |
| stg0 | 35.92 | 43.11 | 187.01 | 84.81 | 13.21 |
| usr0 | 28.06 | 37.36 | 138.28 | 59.58 | 7.49 |
| usr2 | 46.10 | 75.63 | 584.50 | 18.87 | 763.12 |
| wdv0 | 30.78 | 20.42 | 55.84 | 79.92 | 3.18 |
| web0 | 34.56 | 33.35 | 249.67 | 70.12 | 14.91 |
| hm1 | 25.15 | 139.40 | 298.33 | 90.45 | 20.16 |
| hm2 | 10.20 | 73.12 | 77.52 | 98.53 | 2.28 |
| hm3 | 10.21 | 86.28 | 76.11 | 99.86 | 1.74 |
| onl2 | 74.41 | 15.01 | 292.69 | 64.25 | 3.44 |
| Fin1 | 35.92 | 575.94 | 218.59 | 76.84 | 1.08 |
| Fin2 | 24.13 | 76.60 | 159.94 | 17.65 | 1.11 |
| Web1 | 7.46 | 0.95 | 355.38 | 0.02 | 18.37 |
| Web3 | 69.70 | 0.18 | 245.09 | 0.03 | 19.21 |

Table 4.7: Statistics for *part* of I/O workloads we use.

Our trace-driven simulation experiments are conducted based on the spec of real NVMe disks and enterprise I/O workloads. Specifically, we evaluate more than one hundred enterprise workloads from MSR [158], FIU [159] and UMASS [134] trace repositories. These workloads represent applications widely used in real cloud storage systems, such as financial applications, web mail servers, search engines, etc.

Table 4.7 shows the statistics for some of these workloads (out of more than 100 workloads that we are using), where $S$ is the sequential ratio of write I/O (i.e., the ratio between the amount of sequential write I/Os and the amount of total write I/Os), $\lambda$ is the daily logical write rate (GB/day), $P_{Pk}$ is the peak throughput demand with the $5min$ statistical analyses window, $R_W$ is the write I/O ratio (i.e., the ratio between the amount of write I/Os and the amount of total I/Os), and $WSs$ is the working set size (i.e., the spatial capacity demand). The arrival process of these workloads is drawn from an exponential distribution. We use the following metrics to evaluate our MINTCO algorithms: (1) cost per GB during the expected lifetime: the total logical data-averaged TCO during the expected lifetime ($TCO'_{LfPerData}$); (2) resource utilization: the average throughput and space capacity utilization ratios among all disks; and (3) load balancing: the $CV$ of resource utilization ratio across all disks.

**4.2.4.2.2 TCO Experimental Results** We implement both baseline MINTCO and the performance enhanced MINTCO-PERF. Additionally, three versions of MINTCO are considered, such that MINTCO-$v1$ uses the TCO of expected lifetime ($\sum_{i=1}^{N_D}(C_{I_i} + C'_{M_i} \cdot T_{Lf_i})$), MINTCO-$v2$ uses the TCO model of expected lifetime per day ($\frac{\sum_{i=1}^{N_D}(C_{I_i}+C'_{M_i}\cdot T_{Lf_i})}{\sum_{i=1}^{N_D} T_{Lf_i}}$), and MINTCO-$v3$ uses the TCO model of expected lifetime per GB amount ($\frac{\sum_{i=1}^{N_D}(C_{I_i}+C'_{M_i}\cdot T_{Lf_i})}{\sum_{j=1}^{N_W} D_j}$). As expected, none of these baseline MINTCO algorithms consider load balancing and resource utilization during allocation. For comparison, we also implement other widely used allocation algorithms, including $maxRemCycle$ which selects the disk with the greatest number of remaining write cycles, $minWAF$ which chooses the disk with the lowest estimated WAF value *after* adding the newly incoming workload, $minRate$ which chooses the disk with the smallest sum of all its workloads' logical write rates, and $minWorkloadNum$ which selects the disk with the smallest number of workloads.

**(1) minTCO:** We first conduct the experiments running on the disk pool which consists of 20 disks, with 9 different models of NVMe SSDs. In our implementation, we mix about 100 workloads from MSR, FIU, and UMASS with exponentially distributed arrival times in 525 days. Figs. 4.13(a) and (c)

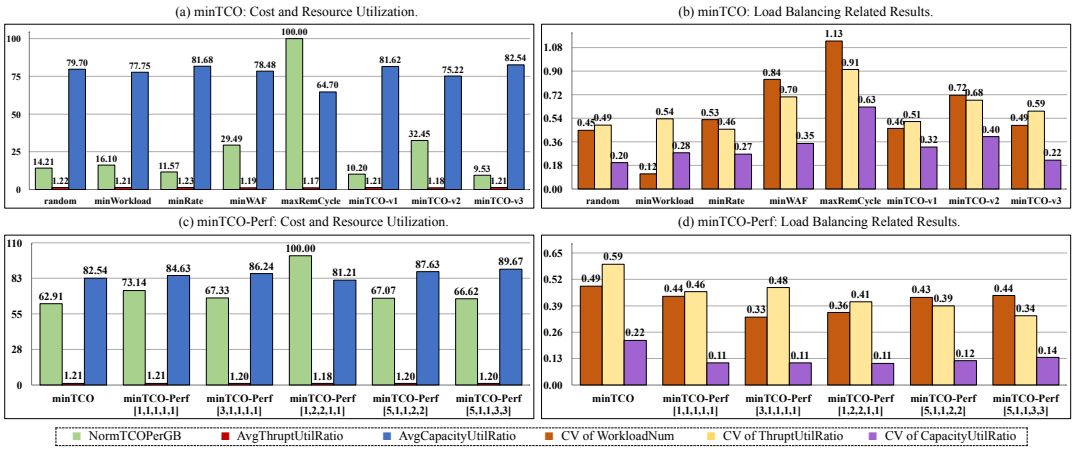



Figure 4.13: TCO rate, resource utilization, and load balancing results under MINTCO and MINTCO-PERF.

show the results of data-avg TCO rates and resource (I/O throughput and space capacity) utilizations under different allocation algorithms. Figs. 4.13(b) and (d) further present the performance of load balancing, e.g., $CV$s of workload number and resource utilizations. First, as shown in Figs. 4.13(a) and (c), MINTCO-v3 achieves the lowest data-avg TCO rate($/GB). We also observe that among the MINTCO family, MINTCO-$v2$ performs the worst, see Fig. 4.13(a), and obtains the largest $CV$s of allocated workload numbers. The reason is that, to some extent, MINTCO-$v2$ aims at maximizing the expected life time by sending almost all workloads to a single disk, in order to avoid "damaging" disks and increasing the TCO.Therefore, it cannot "evenly" allocate the workloads. We further find that $maxRemCycle$ performs the worst among all allocation algorithms, because it does not consider the TCO as well as the varying WAF due to different sequentialities of the running workloads. In summary, $minTCO$-$v3$ is the best choice which considers the expected life time, cost and the expected logical data amount that can be written to each disk.

**(2) minTCO-Perf:** We next implement MINTCO-PERF which is based on MINTCO-$v3$, and considers the data-avg TCO rate as the criterion to choose the disk for the new workload. As described in Sec. 4.2.3.2, MINTCO-PERF uses Eq. 4.12 to find the best candidate under the goal of minimizing TCO, maximizing resource utilization, and balancing the load. There are a set of weight functions (i.e., $f(R_w)$, $g_s(R_r)$, $g_p(R_r)$, $h_s(R_r)$ and $h_p(R_r)$) used in Eq. 4.12 . To investigate the effects of these weight functions, we conduct sensitivity analysis on the average values for the five weight functions in Eq. 4.12. After trying different approaches, and choose the linear function approach to implement weight functions. We show the over-time average value of each function normalized by the minimum function one. For example, "[5,1,1,2,2]" means that all values are

normalized by the second weight function ($g_s(R_r)$). In Figs. 4.13(c) and (g), we observe that space capacity (instead of I/O throughput) is always the system bottleneck (i.e., with high utilization) across different approaches. This is because NVMe SSDs support up to $64K$ I/O queues and up to $64K$ commands per queue (i.e., an aggregation of $2K$ MSI-X interrupts). Meanwhile, workloads we are using here are collected from traditional enterprise servers, which have not been optimized for NVMe's revolutionary throughput capacity improvement. We also find that with a slight sacrifice in TCO, MINTCO-PERF can improve both resource utilization and load balancing. Fig. 4.13(c) further shows that "[5,1,1,3,3]" is the best choice among all cases, which is $3.71\%$ more expensive than the baseline $minTCO$, but increases the space utilization ratio by $7.13\%$, and reduces $CV$ of throughput and space capacity utilization by $0.25$ and $0.8$, respectively. This is because MINTCO-PERF sets TCO and space capacity higher priorities.

### 4.2.5 Summary

We characterize the write amplification of SSDs as a function of the fraction of sequential writes in a workload. We plug this write amplification function into our proposed Total Cost of Ownership (TCO) model, which also considers capital and operational cost, estimated lifetime of flash under different workloads, resource restrictions and performance QoS. Based on the TCO model, we build the online workload allocation algorithm MINTCO and MINTCO-PERF. Experimental results show that MINTCO reduces the ownership cost by up to 90.47%, and MINTCO-PERF further balances the load among disks and maximize the overall resource utilization, while keeping the TCO as low as possible.

# Chapter 5

# Storage Optimization in Big Data Platforms

In the era of big data, huge amount of data are generated by user applications and need to be processed in the datacenter. Besides enhancing the performance and reliability from a hardware perspective (e.g., as we discussed, by utilizing SSDs), we also need to address the limitations induced by the software overheads bottlenecks in big data platforms.

First, we study the existing I/O stack of VM-hypervisor-based platforms which are the infrastructure of big data applications. For historical reasons, current popular deployments of NVMe in VM-hypervisor-based platforms (such as VMware ESXi [17]) have numbers of intermediate queues along the I/O stack. As a result, performance is bottlenecked by synchronization locks in these queues, cross-VM interference induces I/O latency, and most importantly, the up-to-64K-queue capability of NVMe SSDs cannot be fully utilized. We therefore develop a hybrid framework of NVMe-based storage system, called "H-NVMe", which provides two VM I/O stack deployment modes "Parallel Queue Mode" and "Direct Access Mode".

Then, we focus on the intermediate cache optimizing problem in multi-stage data-parallel computing frameworks, such as Apache Spark. Specifically, Spark leverages the distributed memory to cache the intermediate results, i.e., Resilient Distributed Dataset (RDD). By avoiding re-computing cached RDDs, Spark has an huge advantage over other parallel frameworks, especially for processing iterative machine learning applications. To further speed up the execution, Spark also strives to save RDDs as many as possible in the limited memory. By default, LRU is the policy for evicting RDDs when the cache is full. However, when the objective is to minimize total work, LRU is woefully

inadequate, leading to arbitrarily suboptimal caching decisions. In this dissertation, we design an algorithm for the multi-stage big data processing platform to adaptively determine the most valuable intermediate datasets that can be reused in the future to store in the memory. We develop an adaptive algorithm that automates the decision of which RDDs to cache: this amounts to identifying nodes in a Direct Acyclic Graph (DAG) representing computations whose outputs should persist in the memory. Given that the memory is limited, caching algorithms should also determine which RDDs to prioritize, and which ones to evict.

## 5.1 I/O Stack Optimization for Virtualization Hypervisors

### 5.1.1 Motivation

Ever decreasing price per gigabyte of SSDs and their capability of fast operation make them indispensable for large-scale cloud services. They are overtaking HHDs and leaving them far behind by providing orders of magnitude more IOPS and lowering I/O latency. Thanks to the rapid improvement of the process technology, SSDs can possess several chips. Consequently, they may boost up their bandwidth and capacity by obtaining higher degrees of parallelism. With this regards, Non-Volatile Memory Express (NVMe) [37] interface has been introduced to better utilize the parallelism of the emerging storage technologies. However, the limitation induced by software overheads prevents users to perfectly perceive this performance advancement. The overhead of the legacy kernel I/O stack, which has been optimized for slow HDDs, is more noticeable as the storage devices and the connection interfaces get faster. In addition to the new emerging storage technologies which provide high-performance operation through parallelism, container-based virtualization [49] has been a key cloud computing platform which can perfectly take advantages of this parallelism through allowing multiple isolated instances (i.e., containers) of the storage resources. Because of the independence of containers running on top of a single host operating system (i.e., hypervisor), the management of resources through isolation and sharing becomes more important in container-based virtualization. Obviously, having advanced guest kernels is not enough for efficiently sharing system resources. VMware ESXi [17] is one of the most common commercial virtualization platform which can potentially use the NVMe SSDs in the most efficient way by leveraging the massive parallelism and isolation characteristics of NVMe SSDs. Accordingly, VMware ESXi provides VMDK (Virtual Machine Disk) files for each VM, which are completely independent of each other. But, in the case of using NVMe SSDs, they access the storage resource through a single submission and completion

queue in NVMe driver, regardless of high levels of parallelism provided by NVMe. This inefficiency originates from the NVMe driver in the hypervisor, and has become a bottleneck in the storage I/O stack.

In this work, we propose H-NVMe, a novel NVMe framework on VMware ESXi. To best utilize NVMe SSDs, H-NVMe provides two different working modes: "*Parallel Queue Mode*" and "*Direct Access Mode*". In the former working mode, H-NVMe circumvents the built-in *Adapter Queue* of ESXi by emptying it and spreading out its entities between multiple lightweight subqueues in our customized NVMe driver in order to use the parallelism of the device more efficiently. The latter working mode bypasses all the hypervisor queues and directly connects the trusted user application threads to the NVMe *Driver Queue*, to achieve the better performance isolation. H-NVMe can work in either of these two modes in whole or in partial (details see in Sec. 5.1.3). It means that both the parallelism and isolation can be provided at the same time. We evaluate the performance of H-NVMe with a set of representative applications, and the experimental results show that H-NVMe can significantly improve the I/O performance.

### 5.1.2 Background

Storage I/O stack in virtualization environments requires low latency. The main challenges are from (1) the presence of additional software layer such as guest OS; (2) context switching between VM and hypervisor; and (3) queuing delay for I/O operations. These challenges do not cause serious problems and lead to high latency when HDDs are dominating the datacenter [160]. However, in the year of 2017, more and more cloud service vendors started to adopt NVMe SSDs into their storage systems, and for some historical reasons, the current popular deployment of NVMe SSDs in cloud computing hypervisor cannot best utilize the NVMe. For example, Fig. 5.1 depicts a simplified architecture of NVMe I/O stack in VMware's ESXi hypervisor solution (i.e., VMware ESXi Pluggable Storage Architecture) [161]. We can see that there are multiple intermediate queues in the I/O stack. In detail, when an I/O comes to the current original NVMe framework the following steps are needed:

1. *World Queue* is responsible for each VM I/O requests.

2. *Adapter Queue* in the hypervisor layer gathers jobs from *World Queues*.

3. Hypervisor translates the request to `ScsiCommand()` [162] and sends it to the *Device Queue* in the driver.

4. *IOServ* (I/O server) in the Driver acquires an internal completion *lock*.

5. *Driver* sends I/O to the device, waiting for the device to finish the I/O request, while holding the *lock*.

6. Once the I/O is done, *Driver* releases the *lock* and completes the I/O asynchronously when NVMe controller I/O completion interrupt occurs.

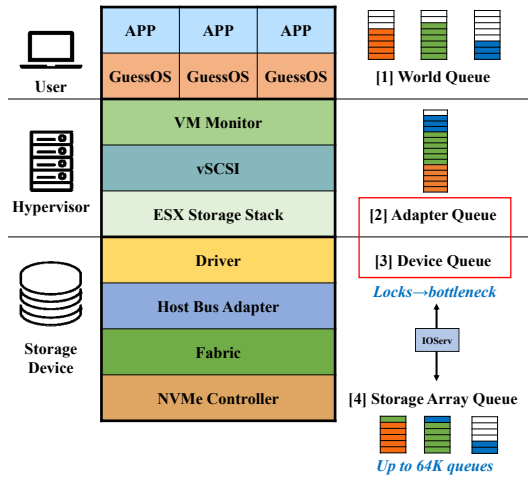



Figure 5.1: Architecture of current NVMe-based VM hypervisor.

This architecture has many issues. Specifically, VMware's PSA layer virtually completes all I/O asynchronously via interrupts. Historically, interrupt driven I/O is efficient for high latency devices such as hard disks, but it also induces a lot of overheads including many context switches. However, it is not efficient for NVMe SSDs. Most importantly, this interrupt driven approach is not lock-free and limits the parallel I/O capacity, since *IOServs* in the upper layer will not send another I/O to the NVMe controller until previous submitted I/O returns pending status. Thus, the NVMe's multiple cores/multiple queues mechanisms are not fully utilized.

To further find the evidence of the disadvantage of having these non-lock-free queues, we conduct a preliminary test where we break down the five locks initialized in the submission and completion queues in the *Adapter Queue*. The statistic results in Fig. 5.2 show that temporal wastes on these locks are not negligible, especially the $lock1$ in submission queue [163] takes near 40% of the total queuing time. As a result, the huge bottleneck between the *Adapter Queue* and the *Device Queue* makes the system not able to best utilize up to 64K queues in the NVMe controller, in addition to the cross-VM interference issues.

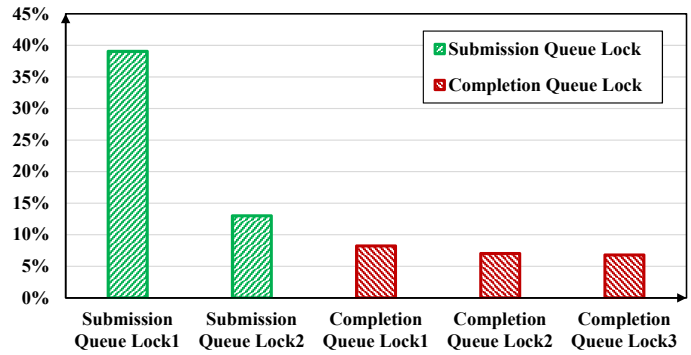



Figure 5.2: Normalized time spent of locks in each queue.

### 5.1.3 Algorithm Design

In this subsection, we propose a hybrid NVMe utilization approach called "H-NVMe" to support parallelism and performance isolation by introducing two modes, namely "*Parallel Queue Mode (PQM)*" and "*Direct Access Mode (DAM)*". The PQM is to enforce the *Adapter Queue* to be empty and use our enhanced subqueues in the driver to speed up the I/O stack. This is relatively straightforward to implement and is a more generalized solution. On the other hand, the DAM can achieve better performance by directly connecting those trusted user application threads to the NVMe driver queues through our customized VAIO (vSphere APIs for IOFiltering) IOFilters [150] attached to their corresponding hosting VMDKs. In contrast, this approach breaks the encapsulation of NVMe resources and thus needs higher permission control processes. Based on SLAs of VMDKs owned by different users, and the security audit requirement (such as attaching IOFilter permission), the cloud manager can select from these two modes for VM deployment in the NVMe-based storage system.

#### 5.1.3.1 Parallel Queue Mode

The challenge of solving the major bottleneck located in the hypervisor layer (observed in Sec. 5.1.1) is that unlike the "*Device Queue*" encapsulated in the driver layer which we have full control, the "*World Queue*" and "*Adapter Queue*" are much engineering-unfriendly and any changes on them will affect the system capability. Thus, our solution is to "force" the *Adapter Queue* to be emptied and forwarded to the driver layer (through some "*fake completion*" signals from the driver), where the enhanced subqueues can help to increase the parallelism and speed up the I/Os, see in the middle column of Fig. 5.3 and Alg. 6.

Unlike the original NVMe design where the single *IOServ* has the "lock" issue, H-NVMe

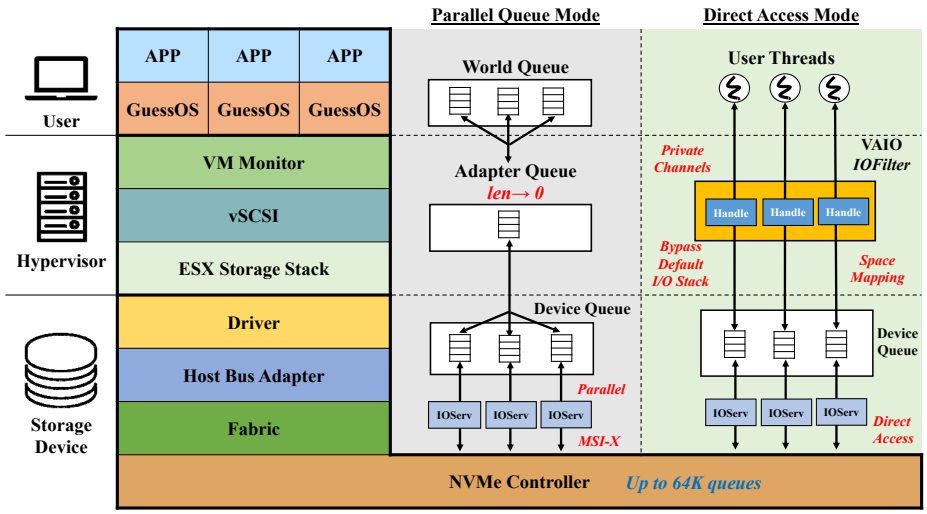



Figure 5.3: I/O paths of two proposed modes.

allows creating more than one *IOServ* threads and subqueues in the "*Device Queue*" of the NVMe driver to handle the I/O with the NVMe controller. Therefore, the lock-free goal can be achieved by *IOServ* threads to only focusing on their own queues, see in Alg. 6 lines 11 to 13. In our implementation, the $callIOServ$ function selects an idle *IOServ* to assign jobs in the round robin order.

Apparently, having more *IOServs* will indeed improve the I/O performance, but it is not free to have infinite *IOServs*, since more CPU and memory resources will be consumed by these threads. As a result, the service rate of each *IOServ* decreases, too. Therefore, the next question that PQM has to address is "*how to dynamically assign subqueue and IOServs number?*", which motivates us to find an adaptive algorithm to automatically select an appropriate number of *IOServs*.

Since the length of *Adapter Queue* is forced to be 0, we can model the problem based on the *M/M/c* queuing theory [164]. Let the total arrival rate (from the *Adapter Queue*) be $\lambda$, and let vector $\vec{\mu}$ denote each *IOServ*'s service rate. This vector has $c$ dimensions, where $c \in [1, c_{max}]$ is the number of *IOServs*, and $c_{max}$ is the preset maximal *IOServ* number. If we increase the number of $c$ (e.g., creating and destroying *IOServs*), each server's service rate will change, and this change can be estimated by a regression function (see in Alg. 6 line 24) based on periodical measurement. Once $\lambda$ and $\vec{\mu}$ are calculated, H-NVMe calls the $optimizeServNum$ function to decide the best number of $c$. The optimization objective of this function is to minimize the total latency. Specifically, different combinations of $c$ and corresponding service rates $\vec{\mu_c}$ are tested. This procedure also uses the $ErlangC$ function [165] to calculate the probability that an arriving job will need to queue (as

opposed to immediately being served). Lastly, to further reduce the cost of changing $c$, we need to limit the frequency of updating $c$ to a preset update epoch window $epochLen$ in Alg. 6 line 22.

**Algorithm 6:** Main Procedures of Parallel Queue Mode.

```
1  Procedure IOSubmissionNVMeHypervisor()
2      while OSLibIOSubmit(ctrlr, vmkCmd, devData) do
3          submissionQueue = getSubmissionQueue(ctrlr);
4          IORequest = getIORequest(ctrlr,vmkCmd);
5          if !IORequest then
6              return VMK_NO_MEMORY;
7          IORequest.vmkCmd = vmkCmd;
8          IORequest.devData = devData;
9          OrigLink = submissionQueue.link;
10         IORequest.link = OrigLink;
11         while !update(submissionQueue.link, OrigLink, IORequest) do
12             OrigLink = submissionQueue.link;
13             IORequest.link = OrigLink;
14         callIOServ(submissionQueue);
15         return VMK_OK;
16 Procedure callIOServ(submissionQueue)
17     curIOServ=selectIOServ();
18     curIOServ.takeJob(submissionQueue);
19     return;
20 Procedure subQueueNum()
21     while True do
22         if curTime MOD epochLen = 0 then
23             λ = updateArrvRate();
24             μ⃗ = regressSubQueueServRate(C_max);
25             c = optimizeServNum(λ, μ⃗);
26 Procedure optimizeServNum(λ, μ⃗)
27     return argmin_{c∈[1,c_max]} [ ErlangC(λ,μ⃗_c) / (cμ⃗_c − λ) + 1/μ⃗_c ];
28 Procedure ErlangC(λ, μ⃗_c)
29     ρ = λ / μ⃗_c;
30     return 1 / (1 + (1−ρ)[c! / (cρ)^c] Σ_{k=0}^{c−1} (cρ)^k / k!);
```

#### 5.1.3.2 Direct Access Mode

Although PQM can improve the queuing performance by moving all jobs from the *Adapter Queue* to our customized multiple subqueues in the driver, it still cannot simplify the complex VMware I/O stack and thus cannot fully utilize the low latency NVMe SSDs and avoid cross-VM interference. To thoroughly reduce the I/O path complexity and support performance isolation, we develop the "*Direct Access Mode*", which allows *trusted* applications whose hosting VMDKs are attached with our customized VAIO IOFilters to bypass the entire VMware I/O stacks and directly use polling I/Os to the NVMe resource shared by multiple users. DAM is transparent to users since it does not require applications to be re-compiled (e.g., linking with a third-party library). This user-space I/O strives to fully utilize the performance of NVMe SSDs while meeting the diverse requirements from user applications and achieving performance isolation.

As illustrated in the third column of Fig. 5.3, in this mode, DAM bypasses the entire VMware I/O stack, and provides *Handles* to grant each *trusted* application in the VM the *root privilege* to manage the access permission of I/O queues in NVMe controller via *Private Channels*. Each user thread is assigned to each application inside VMs. The I/O submissions and completions do not require any driver interventions. As shown in the yellow box, *Handles* will take over all I/O stack of the hypervisor, and map the user space to the NVMe space. Next, the *Device Queue* will be directly assigned to each *Handle*, without being forwarded by the *World Queue* and *Adapter Queue*.

**Algorithm 7:** Main Procedures of Direct Access Mode.

```
1  Procedure IOSubmissionNVMeHypervisor()
2      userLib → kernelDriver ;
3      /* Initialization */ ;
4      if curApp ∈ userLevelWhitelist then
5          new subQueue;
6          new compQueue;
7          subQueue.memoryRegion.map(curApp.memRegion);
8          compQueue.memoryRegion.map(curApp.memRegion);
9          curHandle=new handle(submissionQueue, completionQueue, doorbellReg);
10         curHandleList+=curHandle;
11     forwardIOCmd(curHandleList, NVMeCtrlr);
```

Alg. 7 further explains the detail of DAM. When an application requests a single I/O queue to DAM, H-NVMe checks whether the application (as well as its hosting VM) is allowed to

perform user-level I/Os. If it is in the whitelist (preset based on SLA and security audit requirements), the customized VAIO IOFilter creates a required submission queue and a completion queue (i.e., "*Private Channels*"), see Alg. 7 lines 4-6. H-NVMe then maps their memory regions (including those associated doorbell registers) to the user-space memory region of the application, in lines 7-8. After this initialization process, the application can issue I/O commands directly to the NVMe SSD without any hardware modification or help from the kernel I/O stack, see Alg. 7 lines 9-11.

### 5.1.4 Evaluation

#### 5.1.4.1 Evaluation Methodology

We implement the proposed H-NVMe framework on VMware ESXi hypervisor 6.0.0 [17]. Table 5.1 summarizes our server configuration. We compare the I/O performance of H-NVMe with the original kernel-based I/O with asynchronous I/O support (i.e., *Kernel I/O*) using the Flexible IO Tester (FIO) benchmark [152]. We allocate around 10% of VMs under Direct Access Mode (DAM) to reflect the premium paid rate in enterprise market [166]. Other VMs are deployed under Parallel Queue Mode (PQM). VMDKs of those DAM VMs are attached with a modified IO Filter [150] driver to support the user polling direct access feature. The IO Filter driver is responsible for collecting and forwarding I/O to *Handles* in Fig. 5.3, which thus breaks the encapsulation of the VMware and lets VMs directly operate on the NVMe device.

To compare the I/O performance under different solutions, we evaluate I/O speed results using the average I/O throughputs (IOPS), the bandwidth (MBPS) and the latency for both read and write. To evaluate the cost of different algorithms, we focus on two metrics, i.e., (1) the total runtime of a fixed amount of workload, and (2) the amount of data involved in the context switching.

#### 5.1.4.2 Throughput, Bandwidth and Latency of Mixed Workloads

We run 7 workloads with different average I/O sizes on 7 different VMs hosted by the same hypervisor, configuring the overall sequential ratio of each workloads varies from 30% to 70%. Fig. 5.4 illustrates the average bandwidth and the normalized latency for both read and write I/Os of these workloads under the original NVMe and the proposed H-NVMe frameworks. Generally speaking, H-NVMe achieves the better performance in all subfigures. We also notice that for small block size workloads, H-NVMe performs much better than the original NVMe framework. However, when the workload I/O size increases to 256KB (e.g., the "256KB" bars in Figs. 5.4(a) and (c)), the performance improvement becomes less. This is because larger average I/O sizes have better locality

| Component | Specs |
|---|---|
| Host Server | HPE ProLiant DL380 G9 |
| Host Processor | Intel Xeon CPU E5-2360 v3 |
| Host Processor Speed | 2.40GHz |
| Host Processor Cores | 12 Cores |
| Host Memory Capacity | 64GB DIMM DDR4 |
| Host Memory Data Rate | 2133 MHz |
| Host Hypervisor | VMware ESXi 6.0.0 |
| Storage Device | Samsung NVMe 1725 |
| Form Factor | HHHL |
| Per Disk Size | 3.2TB |
| Max Seq Read Bandwidth | 6,000 MBPS |
| Max Seq Write Bandwidth | 2,000 MBPS |

Table 5.1: Host server configuration.

so that there is less room for the performance improvement. Similarly, once I/O size reaches 256KB, the read latency of the original NVMe is even slightly better, see in Fig. 5.4(b).

#### 5.1.4.3 Temporal and Spatial Overheads of Mixed Workloads

We further investigate the overhead of H-NVMe, using Fig. 5.5 to show both the temporal and spatial overhead of H-NVMe. In detail, from Fig. 5.5(a), we observe that H-NVMe significantly reduces the total runtime for the same amount of workload compared to the original NVMe framework, especially for those workloads with small I/O sizes. The reason is that unlike workloads with larger I/O sizes having high locality even with the original NVMe framework, workloads with small I/O sizes are more beneficial from PQM's parallel processing subqueues.

On the other hand, we evaluate the spatial overhead by measuring the amount of data in the context switching [167]. As shown in Fig. 5.5(b), H-NVMe has slightly higher spatial overheads, because it needs to maintain multiple subqueues in the driver layer which requires few extra context switches. However, the difference of their spatial overheads gets smaller when the average I/O size increases. In fact, the penalty of this extra spatial overhead is much lower than the time saved by H-NVMe. Therefore, we conclude that by eliminating the synchronization lock contention from multiple queues, H-NVMe can significantly reduce the performance overhead of the I/O stack.

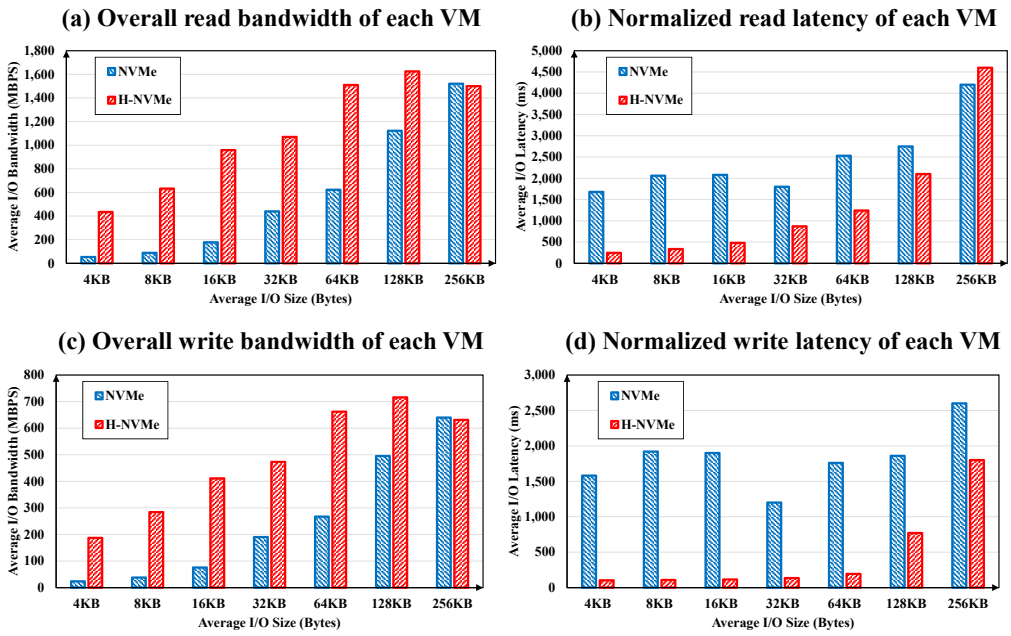



Figure 5.4: Bandwidth and latency results of 7 different workloads under original NVMe and proposed H-NVMe frameworks.

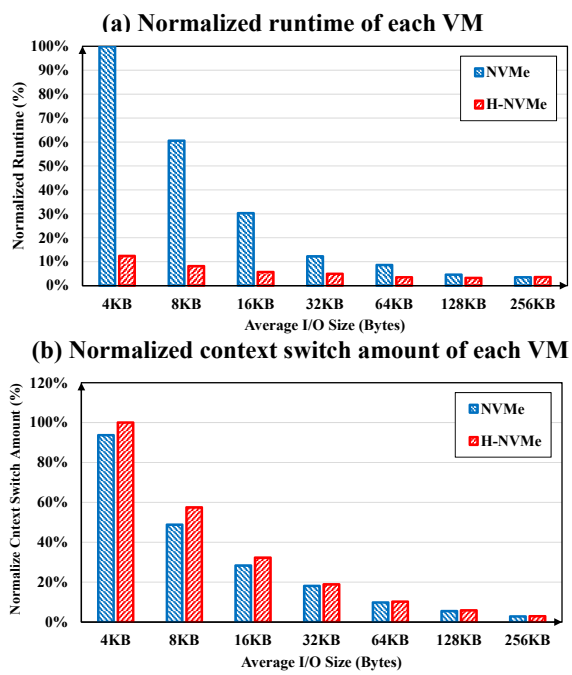



Figure 5.5: Normalized runtime and context switch amount of 7 different workloads under original NVMe and proposed H-NVMe frameworks.

#### 5.1.4.4 Stress Tests on Sequential and Random Workloads

To further investigate the boundary performance improvement brought by H-NVMe under extreme cases, we conduct several stress tests on read- and write-intensive, pure sequential and random workloads in this subsection.

We use the original NVMe's sequential read and write as the benchmark baseline, which reflects the read and write *steady states* (i.e., the "upper bound") under the original NVMe framework. We first present the study on the read-intensive workloads. From Fig. 5.6(a), we observe that H-NVMe has the higher bandwidth in both sequential and random read than the original NVMe upper bound, which validates the effectiveness of H-NVMe. Furthermore, once the average I/O size is greater than 16KB, H-NVMe's sequential read bandwidth is close to the device capacity (i.e., 6,000 MBPS), and meanwhile, the original NVMe cannot reach that point even at the datapoint of 128KB.

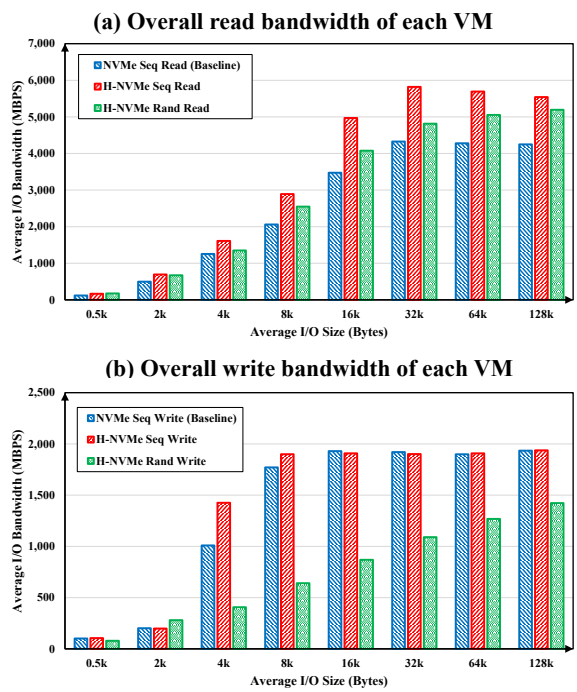



Figure 5.6: Read and write bandwidths of each VM.

We next examine the write-intensive sequential and random workloads. As shown in Fig. 5.6(b), H-NVMe works slightly better than the original NVMe, and the *steady state* is reached after 8KB, where both NVMe and H-NVMe's bandwidths are close to the device capacity (i.e., 2,000 MBPS) for sequential writes. Another observation from Fig. 5.6(b) is that the random write

bandwidth (see the green bar) is increasing linearly with increasing I/O sizes, and thus reduces the gap from the sequential write, which ascribes to H-NVMe's ability to widen the queuing bottlenecks.

### 5.1.5 Summary

We present a hybrid framework called "H-NVMe" to better utilize NVMe SSDs in the modern super-scale cloud computing datacenters. Our work is motivated by the bottleneck caused by multiple intermediate queues in the current deployment of NVMe in the VM-hypervisor environment, which cannot fully utilize the maximum performance throughput (up to 64K queues) of NVMe resources. To solve this bottleneck issue, H-NVMe offers two modes to deploy VMs, i.e., "Parallel Queue Mode" and "Direct Access Mode". The first mode is to increase the parallelism and enable lock-free operations by forwarding jobs from the *Adapter Queue* to our customized enhanced subqueues in the driver. This mode is a generalized solution and is relatively straightforward to implement. The second mode allows trusted applications with VAIO IOFilter attached to their user VMDKs to directly access NVMe SSDs and bypass the entire I/O stack in the hypervisor layer to further ensure performance isolation, which suits premium users who have higher priorities and the permission to attach IOFilter to their VMDKs. We implement H-NVMe on VMware ESXi, and our evaluation results show that the proposed framework outperforms the original NVMe solution under multiple benchmarks.

## 5.2 Intermediate Data Caching Enhancement for Big Data Platforms

### 5.2.1 Motivation

With the rise of big data analytics and cloud computing, cluster-based large-scale data processing has become a common paradigm in many applications and services. Online companies of diverse sizes, ranging from technology giants to smaller startups, routinely store and process data generated by their users and applications on the cloud. Data-parallel computing frameworks, such as Apache Spark [19, 9] and Hadoop [18], are employed to perform such data processing at scale. Jobs executed over such frameworks comprise hundreds or thousands of identical parallel subtasks, operating over massive datasets, and executed concurrently in a cluster environment.

The time and resources necessary to process such massive jobs are immense. Nevertheless, jobs executed in such distributed environments often have significant computational overlaps: different jobs processing the same data may involve common intermediate computations, as illustrated in

Fig. 5.7. Such computational overlaps arise naturally in practice. Indeed, computations performed by companies are often applied to the same data-pipeline: companies collect data generated by their applications and users, and store it in the cloud. Subsequent operations operate over the same pool of data, e.g., user data collected within the past few days or weeks. More importantly, a variety of prominent data mining and machine learning operations involve common preprocessing steps. This includes database projection and selection [168], preprocessing in supervised learning [169], and dimensionality reduction [170], to name a few. Recent data traces from industry have reported $40 \sim 60\%$ recurring jobs in Microsoft production clusters [171], and up to $78\%$ jobs in Cloudera clusters involve data re-access [172].

Exploiting such computational overlaps has a tremendous potential to drastically reduce job computation costs and lead to significant performance improvements. In data-parallel computing frameworks like Spark, computational overlaps inside each job are exploited through caching and *memoization*: the outcomes of computations are stored with the explicit purpose of significantly reducing the cost of subsequent jobs. On the other hand, introducing caching also gives rise to novel challenges in resource management; to that end, the purpose of this dissertation is to design, implement and evaluate caching algorithms over data-parallel cluster computing environments.

Existing data-parallel computing frameworks, such as Spark, incorporate caching capabilities in their framework in a non-automated fashion. The decision of which computation results to cache rests on the developer that submits jobs: the developer explicitly states which results are to be cached, while cache eviction is implemented with the simple policy (e.g., LRU or FIFO); neither caching decisions nor evictions are part of an optimized design. Crucially, determining which outcomes to cache is a hard problem when dealing with jobs that consist of operations with complex dependencies. Indeed, under the Directed Acyclic Graph (DAG) structures illustrated in Fig. 5.7, making caching decisions that minimize, e.g., total work is NP-hard [173, 174]. In detail, jobs to be executed over the cluster arrive at different times $t_1, \ldots, t_5$. Each job is represented by a Directed Acyclic Graph (DAG), whose nodes correspond to operations, e.g., map, reduce, or join, while arrows represent the order of precedence. Crucially, jobs have *computational overlaps*: their DAGs comprise common sets of operations executed over the same data, indicated as subgraphs colored identically across different jobs. Caching such results can significantly reduce computation time.

In this dissertation, we develop an adaptive algorithm for caching in a massively distributed data-parallel cluster computing environment, handling complex and massive data flows. Specifically, a mathematical model is proposed for determining caching decisions that minimize total work, i.e., the total computation cost of a job. Under this mathematical model, we have developed new

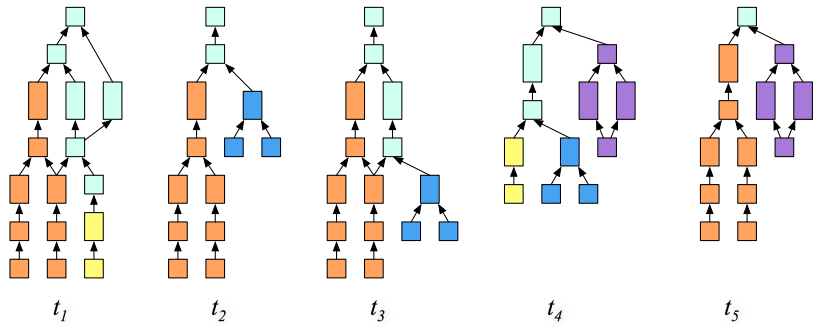



Figure 5.7: Job arrivals with computational overlaps.

*adaptive* caching algorithms to make online caching decisions with optimality guarantees, e.g., minimizing total execution time. Moreover, we extensively validate the performance over several different databases, machine learning, and data mining patterns of traffic, both through simulations and through an implementation over Spark, comparing and assessing their performance with respect to existing popular caching and scheduling policies.

### 5.2.2 Background

#### 5.2.2.1 Resilient Distributed Datasets in Spark

Apache Spark has recently been gaining ground as an alternative for distributed data processing platforms. In contrast to Hadoop and MapReduce [175], Spark is a memory-based general parallel computing framework. It provides *Resilient Distributed Datasets* (RDDs) as a primary abstraction: RDDs are distributed datasets stored in RAM across multiple nodes in the cluster. In Spark, the decision of which RDD to store in the RAM-based cache rests with the developer [176]: the developer explicitly requests for certain results to persist in RAM. Once the RAM cache is full, RDDs are evicted using the LRU policy. Alternatively, developers are further given the option to store evicted RDDs on HDFS, at the additional cost of performing write operations on HDFS. RDDs cached in RAM are stored and retrieved faster; however, cache misses occur either because an RDD is not explicitly cached by the developer, or because it was cached and later evicted. In either case, Spark is resilient to misses at a significant computational overhead: if a requested RDD is neither in RAM nor stored in HDFS, Spark re-computes it from scratch. Overall, cache misses, therefore, incur additional latency, either by reading from HDFS or by fully re-computing the missing RDD.

An example of a job in a data-parallel computing framework like Spark is given in Fig. 5.8. A job is represented as a DAG (sometimes referred to as the *dependency graph*). Each node of the

DAG corresponds to a parallel operation, such as reading a text file and distributing it across the cluster, or performing a map, reduce, or join operation. Edges in the DAG indicate the order of precedence: an operation cannot be executed before all operations pointing towards it are completed, because their outputs are used as inputs for this operation. As in existing frameworks like Spark or Hadoop, the inputs and outputs of operations may be distributed across multiple machines: e.g., the input and output of a map would be an RDD in Spark, or a file partitioned across multiple disks in HDFS in Hadoop.

#### 5.2.2.2 Computational Overlaps

Caching an RDD resulting from a computation step in a job like the one appearing in Fig. 5.8 can have significant computational benefits when jobs may exhibit *computational overlaps*: not only are jobs executed over the same data, but also consist of operations that are repeated across multiple jobs. In detail, an example of a parallel job represented as a DAG as shown in Fig. 5.8. Each node corresponds to an operation resulting RDD that can be executed over a parallel cluster (e.g., a map, reduce, or join operation). DAG edges indicate precedence. Simple, crunodes (in/out) and cross nodes are represented with solid or lined textures. This is also illustrated in Fig. 5.7: jobs may be distinct, as they comprise different sets of operations, but certain subsets of operations (shown as identically colored subgraphs in the DAG of Fig. 5.7) are the same, i.e., execute the same primitives (maps, joins, etc.) and operate over the same data.

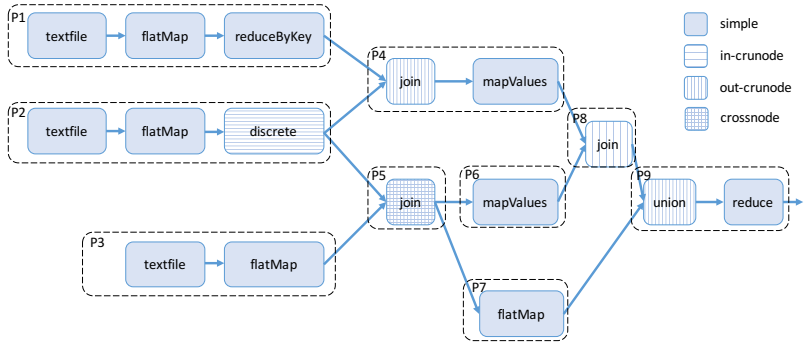



Figure 5.8: Job DAG example.

Computational overlaps arise in practice for two reasons. The first is that operations performed by companies are often applied to the same data-pipeline: companies collect data generated by their applications and users, which they maintain in the cloud, either directly on a distributed file system like HDFS, or on NoSQL databases (like Google's Datastore [177] or Apache HBase [178]).

Operations are therefore performed on the same source of information: the latest data collected within a recent period of time. The second reason for computational overlaps is the abundance of commonalities among computational tasks in data-parallel processing. Commonalities occur in several classic data-mining and machine learning operations heavily utilized in inference and prediction tasks (such as predictions of clickthrough rates and user profiling). We give some illustrative examples below:

**Projection and Selection:** The simplest common preprocessing steps are *projection* and *selection* [168]. For example, computing the mean of a variable `age` among tuples satisfying the predicate $\texttt{gender} = \texttt{female}$ and $\texttt{gender} = \texttt{female} \wedge \texttt{income} \geq 50\text{K}$ might both first reduce a dataset by selecting rows in which $\texttt{gender} = \texttt{female}$. Even in the absence of a relational database, as in the settings we study here, projection (i.e., maintaining only certain feature columns) and selection (i.e., maintaining only rows that satisfy a predicate) are common. For example, building a classifier that predicts whether a user would click on an advertisement relies upon first restricting a dataset containing all users to the history of the user's past clicking behavior. This is the same irrespective of the advertisement for which the classifier is trained.

**Supervised Learning:** Supervised learning tasks such as regression and classification [169], i.e., training a model from features for the purpose of predicting a label (e.g., whether a user will click on an advertisement or image) often involve common operations that are label-independent. For example, performing ridge regression first requires computing the co-variance of the features [169], an identical task irrespective of the label to be regressed. Similarly, kernel-based methods like support vector machines require pre-computing a kernel function across points, a task that again remains the same irrespective of the labels to be regressed [179]. Using either method to, e.g., regress the click-through rate of an ad, would involve the same preprocessing steps, irrespectively of the labels (i.e., clicks pertaining to a specific ad) being regressed.

**Dimensionality Reduction:** Preprocessing also appears in the form of *dimensionality reduction*: this is a common preprocessing step in a broad array of machine learning and data mining tasks, including regression, classification, and clustering. Prior to any such tasks, data is first projected in a lower dimensional space that preserves, e.g., data distances. There are several approaches to doing this, including principal component analysis [180], compressive sensing [170], and training autoregressive neural networks [181], to name a few. In all these examples, the same projection would be performed on the data prior to subsequent processing, and be reused in the different tasks described above.

To sum up, the presence of computational overlaps across jobs gives rise to a tremendous

opportunity of reducing computational costs. Such overlaps can be exploited precisely through the caching functionality of a data-parallel framework. If a node in a job is cached (i.e., results are *memoized*), then neither itself nor any of its predecessors need to be re-computed.

#### 5.2.2.3 Problems and Challenges

Designing caching schemes poses several significant challenges. To begin with, making caching decisions is an inherently combinatorial problem. Given (i) a storage capacity constraint, (ii) a set of jobs to be executed, (iii) the size of each computation result, and (iv) a simple linear utility on each job, the problem is reduced to a knapsack problem, which is NP-hard. The more general objectives we discussed above also lead to NP-hard optimization problems [182]. Beyond this inherent problem complexity, even if jobs are selected from a pool of known jobs (e.g., classification, regression, and querying), the sequence to submit jobs within a given time interval *is a priori unknown*. The same may be true about statistics about upcoming jobs, such as the frequency with which they are requested. To that end, a practical caching algorithm must operate in an *adaptive* fashion: it needs to make online decisions on what to cache as new jobs arrive, and adapt to changes in job frequencies. In Spark, LRU is the default policy for evicting RDDs when the cache is full. There are some other conventional caching algorithms such as LRU variant [52] that maintains the most recent accessed data for future reuse, and ARC [183] and LRFU [184] that consider both frequency and recency in the eviction decisions. When the objective is to minimize total work, these conventional caching algorithms are woefully inadequate, leading to arbitrarily suboptimal caching decisions [173]. Recently, a heuristic policy [102], named "Least Cost Strategy" (LCS), was proposed to make eviction decisions based on the recovery temporal costs of RDDs. However, this is a heuristic approach and again comes with no guarantees. In contrast, we intend to leverage Spark's internal caching mechanism to implement our caching algorithms and deploy and evaluate them over the Spark platform, while also attaining formal guarantees.

### 5.2.3 Algorithm Design

In this subsection, we introduce a formal mathematical model for making caching decisions that minimize the expected total work, i.e., the total expected computational cost for completing all jobs. The corresponding caching problem is NP-hard, even in an offline setting where the popularity of jobs submitted to the cluster is *a priori known*. Nevertheless, we show it is possible to pose this optimization problem as a submodular maximization problem subject to knapsack constraints.

This allows us to produce a $1 - 1/e$ approximation algorithm for its solution. Crucially, when job popularity is *not known*, we have devised an adaptive algorithm for determining caching decisions probabilistically, that makes caching decisions lie within $1 - 1/e$ approximation from the offline optimal, in expectation.

#### 5.2.3.1 DAG/Job Terminology

We first introduce the terminology we use in describing caching algorithms. Consider a job represented as a DAG as shown in Fig. 5.8. We denote $J_x.S_y$ as stage $y$ in job $x$, then we have $J_0.S_0 = J_1.S_1 = J_2.S_0 = J_3.S_1$, $J_1.S_{0\sim5} = J_3.S_{0\sim5}$, and $J_0.S_{0\sim1} = J_2.S_{0\sim1}$. Unfortunately, even sharing the same computational overlap, by default these subgraphs will be assigned with different stage/RDD IDs by Spark since they are from different jobs.

Let $G(V, E)$ be the graph representing this DAG, whose nodes are denoted by $V$ and edges are denoted by $E$. Each node is associated with an operation to be performed on its inputs (e.g., map, reduce, join, etc.). These operations come from a well-defined set of operation primitives (e.g., the operations defined in Spark). For each node $v$, we denote as $\texttt{op}(v)$ the operation that $v \in V$ represents. The DAG $G$ as well as the labels $\{\texttt{op}(v), v \in V\}$ fully determine the job. A node $v \in V$ is a *source* if it contains no incoming edges, and a *sink* if it contains no outgoing edges. Source nodes naturally correspond to operations performed on "inputs" of a job (e.g., reading a file from the hard disk), while sinks correspond to "outputs". Given two nodes $u, v \in V$, we say that $u$ is a *parent* of $v$, and that $v$ is a *child* of $u$, if $(u, v) \in E$. We similarly define *predecessor* and *successor* as the transitive closures of these relationships. For $v \in V$, we denote by $\texttt{pred}(v) \subset V$, $\texttt{succ}(c) \subset V$ the sets of predecessors and successors of $v$, respectively. Note that the parent/child relationship is the opposite to usually encountered in trees, where edges are usually thought of as pointing away from the root/sink towards the leaves/sources. We call a DAG a *directed tree* if (i) it contains a unique sink, and (ii) its undirected version (i.e., ignoring directions) is acyclic.

#### 5.2.3.2 Mathematical Model

Consider a setting in which all jobs are applied to the same dataset; this is without loss of generality, as multiple datasets can be represented as a single dataset–namely, their union–and subsequently adding appropriate projection or selection operations as preprocessing to each job. Assume further that each DAG is a directed tree. Under these assumptions, let $\mathcal{G}$ be the set of all possible jobs that can operate on the dataset. We assume that jobs $G \in \mathcal{G}$ arrive according to a

stochastic stationary process with rate $\lambda_G > 0$. Recall that each job $G(V, E)$ comprises a set of nodes $V$, and that each node $v \in V$ corresponds to an operation $\texttt{op}(v)$. We denote by as $c_v \in \mathbb{R}_+$ the time that it takes to execute this operation given the outputs of its parents, and $s_v \in \mathbb{R}_+$ be the size of the output of $\texttt{op}(v)$, e.g., in Kbytes. Without caching, the *total-work* of a job $G$ is then given by $W(G(V, E)) = \sum_{v \in V} c_v$. We define the *expected total work* as:

$$\bar{W} = \sum_{G \in \mathcal{G}} \lambda_G \cdot W(G) = \sum_{G(V,E) \in \mathcal{G}} \lambda_{G(V,E)} \sum_{v \in V} c_v. \tag{5.1}$$

We say that two nodes $u, u'$ are *identical*, and write $u = u'$, if both these nodes and all their predecessors involve exactly the same operations. We denote by $\mathcal{V}$ the union of all nodes of DAGs in $\mathcal{G}$. A *caching strategy* is a vector $x = [x_v]_{v \in \mathcal{V}} \in \{0, 1\}^{|\mathcal{V}|}$, where $x_v \in \{0, 1\}$ is a binary variable indicating whether we have cached the outcome of node $v$ or not. As jobs in $\mathcal{G}$ are directed trees, when node $v$ is cached, *there is no need to compute that node or any predecessor of that node*. Hence, under a caching strategy $x$, the total work of a job $G$ becomes:

$$W = \textstyle\sum_{v \in V} c_v (1 - x_v) \prod_{u \in \texttt{succ}(v)} (1 - x_u). \tag{5.2}$$

Intuitively, this states that the cost $c_v$ of computing $\texttt{op}(v)$ needs to be paid if and only if *neither* $v$ *nor* any of its successors have been cached.

#### 5.2.3.3 Maximizing the Caching Gain: Offline Optimization

Given a cache of size $K$ Kbytes, we aim to solve the following optimization problem:

MAXCACHINGGAIN

Maximize:
$$F(x) = \bar{W} - \sum_{G \in \mathcal{G}} \lambda_G W(G, x) \tag{5.3a}$$
$$= \sum_{G(V,E) \in \mathcal{G}} \lambda_G \sum_{v \in V} c_v \big[1 - (1 - x_v) \prod_{u \in \texttt{succ}(v)} (1 - x_u)\big] \tag{5.3b}$$

Subject to:
$$\textstyle\sum_{v \in \mathcal{V}} s_v x_v \leq K, \quad x_v \in \{0, 1\}, \text{ for all } v \in \mathcal{V}. \tag{5.3c}$$

Following [173], we call function $F(x)$ the *caching gain*: this is the reduction on total work due to caching. This offline problem is NP-hard [174]. Seen as an objective over the set of nodes $v \in \mathcal{V}$ cached, $F$ is a *monotone, submodular* function. Hence, Eq. (5.3) is a submodular maximization problem with a knapsack constraint. When all outputs have the same size, the classic greedy algorithm by Nemhauser et al. [185] yields a $1 - 1/e$ approximation. In the case of general knapsack constraints, there exist well-known modifications of the greedy algorithm that yields the same approximation ratio [186, 187, 188].

Beyond the above generic approximation algorithms for maximizing submodular functions, Eq. (5.3) can be solved by *pipage rounding* [189]. In particular, there exists a concave function $L : [0,1]^{|\mathcal{V}|}$ such that:

$$(1 - 1/e)L(x) \leq F(x) \leq L(x), \quad \text{for all } x \in [0,1]^{|\mathcal{V}|}. \tag{5.4}$$

This *concave relaxation* of $F$ is given by:

$$L(x) = \sum_{G(V,E)\in\mathcal{G}} \lambda_G \sum_{v\in V} c_v \min\big\{1, x_v + \sum_{u\in\texttt{succ}(v)} x_u\big\}. \tag{5.5}$$

Pipage rounding solves Eq. (5.3) by replacing objective $F(x)$ with its concave approximation $L(x)$ and relaxing the integrality constraints Eq. (5.3c) to the convex constraints $x \in [0,1]^{|\mathcal{V}|}$. The resulting optimization problem is convex–in fact, it can be reduced to a linear program, and thus solved in linear time. Having solved this convex optimization problem, the resulting fractional solution is subsequently rounded to produce an integral solution. Several polynomial time rounding algorithms exist (see, e.g., [189, 190], and [188] for knapsack constraints). Due to Eq. (5.4) and the specific design of the rounding scheme, the resulting integral solution is guaranteed to be within a constant approximation of the optimal [189, 188].

#### 5.2.3.4 An Adaptive Algorithm with Optimality Guarantees

As discussed above, if the arrival rates $\lambda_G$, $G \in \mathcal{G}$, are known, we can determine a caching policy within a constant approximation from the optimal solution to the (offline) problem MAXCACHINGGAIN by solving a convex optimization problem. In practice, however, the arrival rates $\lambda_G$ may *not* be known. To that end, we are interested in an *adaptive* algorithm, that converges to caching decisions *without any prior knowledge of job arrival rates* $\lambda_G$. Building on [173], we propose an adaptive algorithm for precisely this purpose. We describe the details of this adaptive algorithm in Sec. 5.2.4.1. In short, our adaptive algorithm performs *projected gradient ascent* over concave function $L$, given by Eq. (5.5). That is, our algorithm maintains at each time a fractional $y \in [0,1]^{|\mathcal{V}|}$, capturing the probability with which each RDD should be placed in the cache. Our algorithm collects information from executed jobs; this information is used to produce an estimate of the gradient $\nabla L(y)$. In turn, this is used to adapt the probabilities $y$ that we store different outcomes. Based on these adapted probabilities, we construct a randomized placement $x$ satisfying the capacity constraint Eq. (5.3c). We can then show that the resulting randomized placement has the following property:

**Algorithm 8:** A heuristic caching algorithm.

```
1  Procedure processJobs(𝒢)
2      C_𝒢 = Historical RDD access record;
3      C_G = Current job RDD access record;
4      for G ∈ 𝒢 do
5          processJob(G(V, E), C_G);
6          updateCache(C_G, C_𝒢);
7  Procedure processJob(G(V, E), C)
8      C_G.clear();
9      for v∈V do
10         v.accessed=False;
11         toAccess=set(DAG.sink());
12         while toAccess≠ ∅ do
13             v=toAccess.pop();
14             C_G[v]=estimateCost(v);
15             if not v.cached then
16                 for u ∈ v.parents do
17                     if not u.accessed then
18                         toAccess.add(u);
19             access(v); /* Iterate RDD v. */
20             v.accessed=True;
21     return;
22 Procedure estimateCost(v)
23     cost=compCost[v]; /* If all parents are ready. */
24     toCompute=v.parents /* Check each parent. */
25     while toCompute ≠ ∅ do
26         u=toCompute.pop();
27         if not (u.cached or u.accessed or u.accessedInEstCost) then
28             cost+=compCost[u];
29             toCompute.appendList(u.parents);
30             u.accessedInEstCost=True;
31     return cost;
32 Procedure updateCache(C_G, C_𝒢)
33     for v ∈ C_𝒢 do
34         if v ∈ C_G then
35             C_𝒢[v] = (1 − β) × C_𝒢[v] + β × C_G[v];
36         else
37             C_𝒢[v] = (1 − β) × C_𝒢[v];
38         updateCacheByScore(C_𝒢);
39     return;
```

**Theorem 1** *If $x(t)$ is the placement at time $t$, then $\lim_{t\to\infty} \mathbb{E}[F(x(t))] \geq \big(1 - 1/e\big)F(x^*)$, where $x^*$ is an optimal solution to the offline problem* MAXCACHINGGAIN *(Eq.* (5.3)*).*

The proof of Thm. 1 can be found in Sec. 5.2.4.3.

#### 5.2.3.5 A Heuristic Adaptive Algorithm

Beyond attaining such guarantees, our adaptive algorithm gives us a great intuition to prioritize computational outcomes. Indeed, the algorithm prioritizes nodes $v$ that have a high gradient component $\partial L/\partial x_v$ and a low size $s_v$. Given a present placement, RDDs should enter the cache if they have a high value w.r.t. the following quantity (Sec. 5.2.4.3):

$$\tfrac{\partial L}{\partial x_v}/s_v \simeq \left(\textstyle\sum_{G\in\mathcal{G}:v\in G} \lambda_G \times \Delta(w)\right)/s_v, \tag{5.6}$$

where $\Delta(w)$ is the difference in total work if $v$ is not cached. This intuition is invaluable in coming up with useful heuristic algorithms for determining what to place in a cache. In contrast to, e.g., LRU and LFU, that prioritize jobs with high request rate, Eq. (5.6) suggests that a computation should be cached if (i) it is requested often, (ii) caching it can lead to a significant reduction on the total work, and (iii) it has small size. Note that (ii) is *dependent on other caching decisions made by our algorithm*. Observations (i), (ii), and (iii) are intuitive, and the specific product form in Eq. (5.6) is directly motivated and justified by our formal analysis. They give rise to the following simple heuristic adaptive algorithm: for each job submitted, maintain a moving average of (i) the request rate of individual nodes it comprises, and (ii) the cost that one would experience if these nodes are not cached. Then, place in the cache only jobs that have a high such value, when scaled by the size $s_v$.

Alg. 8 shows the main steps of our heuristic adaptive algorithm. It updates the cache (i.e., storage memory pool) after the execution of each job (line 5) based on decisions made in the $updateCache$ function (line 6), which considers both the historical (i.e., $C_{\mathcal{G}}$) and current RDD (i.e., $C_G$) cost scores. Particularly, when iterating RDDs in each job following a recursive fashion, an auxiliary function $estimateCost$ is called to calculate and record the temporal and spatial cost of each RDD in that job (see line 14 and lines 22 to 31). Notice that $estimateCost$ does not actually access any RDDs, but conducts DAG-level analysis for cost estimation which will be used to determine cache contents in the $updateCache$ function. In addition, a hash mapping table is also used to record and detect computational overlap cross jobs (details see in our implementation in Sec. 5.2.5.3). After that, we iterate over each RDD's parent(s) (lines 16 to 18). Once all its parent(s)

is(are) ready, we access (i.e., compute) the RDD (line 19). Lastly, the $updateCache$ function first updates the costs of all accessed RDDs to decide the quantities cost collected above with a moving average window using a decay rate of $\beta$, or using an Exponentially Weighted Moving Average (EWMA) [191]. Next, $updateCache$ makes cross-job cache decisions based on the sorting results of the moving average window by calling the $updateCacheByScore$ function. The implementation of this function can (i) refresh the entire RAM by top score RDDs; or (ii) evict lower score old RDDs to insert higher score new RDDs.

### 5.2.4 Algorithm Correctness Proof

#### 5.2.4.1 Online Algorithm Overview

We hereby describe our adaptive algorithm for solving MAXCACHINGGAIN without a prior knowledge of the demands $\lambda_G$, $G \in \mathcal{G}$. The algorithm is based on [173], which solves a problem with a similar objective, but with matroid (rather than knapsack) constraints. We depart from [173] in both the objective studied–namely, Eq. (5.3a)– as well as in the rounding scheme used: the presence of knapsack constraints implies that a different methodology needs to be applied to round the fractional solution produced by the algorithm in each step.

We partition time into periods of equal length $T > 0$, during which we collect access statistics for different RDDs. In addition, we maintain as state information the *marginals* $y_v \in [0,1]^{|\mathcal{V}|}$: intuitively each $y_v$ captures the probability that node $v \in \mathcal{V}$ is cached. When the period ends, we (i) adapt the state vector $y = [y_v]_{v\in\mathcal{V}} \in [0,1]^{|\mathcal{V}|}$, and (ii) reshuffle the contents of the cache, in a manner we describe below.

**State Adaptation:** We use RDD access and cost measurements collected during a period to produce a random vector $z = [z_v]_{v\in\mathcal{V}} \in \mathbb{R}_+^{|\mathcal{V}|}$ that is an unbiased estimator of a subgradient of $L$ w.r.t. to $y$. That is, if $y^{(k)} \in [0,1]^{|\mathcal{V}|}$ is the vector of marginals at the $k$-th measurement period, $z = z(y^{(k)})$ is a random variable satisfying:

$$\mathbb{E}\big[z(y^{(k)})\big] \in \partial L(y^{(k)}) \tag{5.7}$$

where $\partial L(y)$ is the set of subgradients of $L$ w.r.t $y$. We specify how to produce such estimates below, in Sec. 5.2.4.2.

Having these estimates, we adapt the state vector $y$ as follows: at the conclusion of the $k$-th period, the new state is computed as

$$y^{(k+1)} \leftarrow \mathcal{P}_{\mathcal{D}}\left(y^{(k)} + \gamma^{(k)} \cdot z(y^{(k)})\right), \tag{5.8}$$

where $\gamma^{(k)} > 0$ is a gain factor and $\mathcal{P}_{\mathcal{D}}$ is the projection to the set of relaxed constraints:

$$\mathcal{D} = \left\{ y \in [0,1]^{|\mathcal{V}|} : \sum_{v \in \mathcal{V}} s_v y_v = K \right\}.$$

Note that $\mathcal{P}_{\mathcal{D}}$ is a projection to a convex polytope, and can thus be computed in polynomial time.

**State Smoothening:** Upon performing the state adaptation Eq. (5.8), each node $v \in V$ computes the following "sliding average" of its current and past states:

$$\bar{y}^{(k)} = \textstyle\sum_{\ell=\lfloor \frac{k}{2} \rfloor}^{k} \gamma^{(\ell)} y^{(\ell)} / \left[ \sum_{\ell=\lfloor \frac{k}{2} \rfloor}^{k} \gamma^{(\ell)} \right]. \tag{5.9}$$

This "state smoothening" is necessary precisely because of the non-differentiability of $L$ [192]. Note that $\bar{y}^{(k)} \in \mathcal{D}$, as a convex combination of points in $\mathcal{D}$.

**Cache Placement:** Finally, at the conclusion of a timeslot, the smoothened marginals $\bar{y}^{(k)} \in [0,1]^{|\mathcal{V}|}$ are *rounded*, to produce a new integral placement $x^{(k)} \in \{0,1\}^{|\mathcal{V}|}$ that satisfies the knapsack constraint Eq. (5.3c). There are several ways of producing such a rounding [189, 190, 188]. We follow the probabilistic rounding of [188] (see also [193]): starting from a fractional $y$ that maximizes $L$ over $\mathcal{D}$, the resulting (random) integral $x$ is guaranteed to be within $1 - 1/e$ from the optimal, in expectation.

#### 5.2.4.2 Constructing an Unbiased Estimator of $\partial L(y)$.

To conclude our algorithm description, we outline here how to compute the unbiased estimates $z$ of the subgradients $\partial L(y(k))$ during a measurement period. In the exposition below, drop the superscript $\cdot^{(k)}$ for brevity.

The estimation proceeds as follows.

1. Every time a job $G(V, E)$ is submitted for computation, we compute the quantity

$$t_v = \textstyle\sum_{v \in V} c_v \mathbb{1}\big(x_v + \sum_{u \in \mathtt{succ}(v)} x_u \leq 1\big),$$

   where

$$\mathbb{1}(A) = \begin{cases} 1, & \text{if } A \text{ is true,} \\ 0, & \text{o.w.} \end{cases}$$

2. Let $\mathcal{T}_v$ be the set of quantities collected in this way at node $v$ regarding item $v \in \mathcal{V}$ during a measurement period of duration $T$. At the end of the measurement period, we produce the following estimates:

$$z_v = \textstyle\sum_{t \in \mathcal{T}_v} t/T, \quad v \in \mathcal{V}. \tag{5.10}$$

Note that, in practice, $z_v$ needs to be computed only for RDDs $v \in \mathcal{V}$ that have been involved in the computation of some job in the duration of the measurement period.

It is easy to show that the above estimate is an unbiased estimator of the subgradient:

**Lemma 1** *For $z = [z_v]_{v\in\mathcal{V}} \in \mathbb{R}_+^{|\mathcal{V}|}$ the vector constructed through coordinates Eq.* (5.10)*,*

$$\mathbb{E}[z(y)] \in \partial L(y) \text{ and } \mathbb{E}[\|z\|_2^2] < C^2|\mathcal{V}|^2(\Lambda^2 + \frac{\Lambda}{T}),$$

*where $C = \max_{v\in\mathcal{V}} c_v$ and $\Lambda = \max_{v\in\mathcal{V}} \sum_{G(V,E)\in\mathcal{G}:v\in V} \lambda_G$.*

The proof of this lemma is almost identical, *mutatis mutandis*, to the proof of Lemma 1 in [173].

#### 5.2.4.3 Proof of Thm. 1.

To prove Thm. 1, we first have that, by setting $\gamma^{(k)} = \Theta(1/\sqrt{k})$,

$$\lim_{k\to\infty} \mathbb{E}[L(\bar{y}^k)] = \max_{y\in\mathcal{D}} L(y), \tag{5.11}$$

and, in particular,

$$\lim_{k\to\infty} \mathbf{P}[\bar{y}^{(k)} \in \operatorname*{argmax}_{y\in\mathcal{D}} L(y)] = 1. \tag{5.12}$$

Eq. (5.11) is a consequence of Lemma 1 and Thm. 14.1.1, page 215 of Nemirofski [192]. On the other hand, Eq. (5.4) implies that

$$F(\bar{y}^{(k)}) \geq (1 - 1/e) \max_{y\in\mathcal{D}} F(y) \tag{5.13}$$

for every $\bar{y}^{(k)} \in \operatorname{argmax}_{y\in\mathcal{D}} L(y)$, while the rounding scheme of [188] ensures that the same property is attained by the rounded $x^{(k)}$ in expectation. This, along with Eq. (5.12), implies the theorem.

### 5.2.5 Evaluation

In this subsection, we first demonstrate the performance of our adaptive caching algorithm (Alg. 8) on a simple illustrative example. We then build a simulator to analyze the performance of large scale synthetic traces with complex DAGs. Lastly, we validate the effectiveness of our adaptive algorithm by conducting real experiments in Apache Spark with real-world machine learning workloads.

#### 5.2.5.1 Numerical Analysis

We use a simple example to illustrate how our adaptive algorithm (i.e., Alg. 8) performs w.r.t minimizing total work. This example is specifically designed to illustrate that our algorithm significantly outperforms the default LRU policy used in Spark. Assume that we have 5 jobs ($J_0$ to $J_4$) each consisting of 3 RDDs, the first 2 of which are common across jobs. That is, $J_0$'s DAG is $R_0 \rightarrow R_1 \rightarrow R_2$, $J_1$ is $R_0 \rightarrow R_1 \rightarrow R_3$, $J_2$ is $R_0 \rightarrow R_1 \rightarrow R_4$, etc. The calculation time for $R_1$ is 100 seconds while the calculation time for other RDDs (e.g., $R_2$, $R_3$,...) is 10 seconds. We submit this sequence of jobs twice, with the interarrival time of 10 seconds between jobs. Thus, we have 10 jobs in a sequence of $\{J_0, J_1, ..., J_4, J_0, J_1, ..., J_4\}$. We set the size of each RDD as 500MB and the cache capacity as 500MB as well. Hence, at most one RDD can be cached at any moment.

Table 5.2 shows the experimental results of this simple example under LRU and our algorithm. Obviously, LRU cannot well utilize the cache because the recently cached RDD (e.g., $R_2$) is always evicted by the newly accessed RDD (e.g., $R_3$). As a result, none of the RDDs are hit under the LRU policy. By producing an estimation of the gradient on RDD computation costs, our algorithm instead places $R_1$ in the cache after the second job finishes and thus achieves a higher hit ratio of 36%, i.e., 8 out of 22 RDDs are hit. The "total work" (i.e., the total calculation time for finishing all jobs) is significantly reduced as well under our algorithm.

| **Policy** | $\mathbf{J_0}$ | $\mathbf{J_1}$ | $\mathbf{J_2}$ | $\mathbf{J_3}$ | ... | $\mathbf{J_4}$ | **Hit Ratio** | **Total Work** |
|---|---|---|---|---|---|---|---|---|
| LRU | $R_2$ | $R_3$ | $R_4$ | $R_5$ | ... | $R_6$ | 0.0% | 1100 |
| Adaptive | $R_2$ | $R_1$ | $R_1$ | $R_1$ | ... | $R_1$ | 36.4% | 300 |

Table 5.2: Caching results of the simple case.

#### 5.2.5.2 Simulation Analysis

To further validate the effectiveness of our proposed algorithm, we scale up our synthetic trace by randomly generating a sequence of 1000 jobs to represent real data analysis applications with complex DAGs. Fig. 5.9 shows an example of some jobs' DAGs from our synthetic trace, where some jobs include stages and RDDs with the same generating logic chain. For example, stage 0 in $J_0$ and stage 1 in $J_1$ are identical, but their RDD IDs are different and will be computed twice. On average, each of these jobs consists of six stages and each stage has six RDDs. The average RDD size is 50MB. We use a decay rate of $\beta = 0.6$.

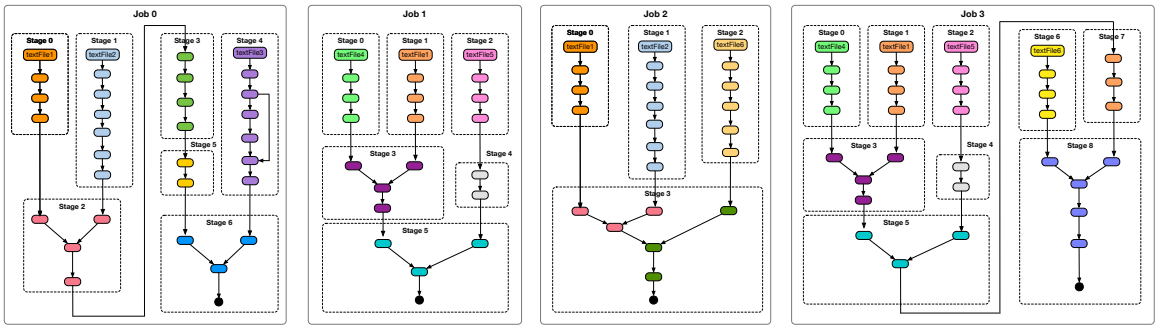



Figure 5.9: An example of RDD dependency in synthetic jobs.

We implement four caching algorithms for comparison: (1) NoCache: a baseline policy, which forces Spark to ignore all user-defined *cache/persist* demands, and thus provides the lower bound of caching performance; (2) LRU: the default policy used in Spark, which evicts the least recent used RDDs; (3) FIFO: a traditional policy which evicts the earliest RDD in the RAM; and (4) LCS: a recently proposed policy, called "Least Cost Strategy" [102], which uses a heuristic approach to calculate each RDD's recovery temporal cost to make eviction decisions. The main metrics include: (a) *RDD hit ratio* that is calculated as the ratio between the number of RDDs hit in the cache and the total number of accessed RDDs, or the ratio between the size of RDDs hit in the cache and the total size of accessed RDDs; (b) *Number of accessed RDDs* and *total amount of accessed RDD data size* that need to be accessed through the experiment; (c) *Total work* (i.e., makespan) that is the total calculation time for finishing all jobs; and (d) *Average waiting time* for each job.

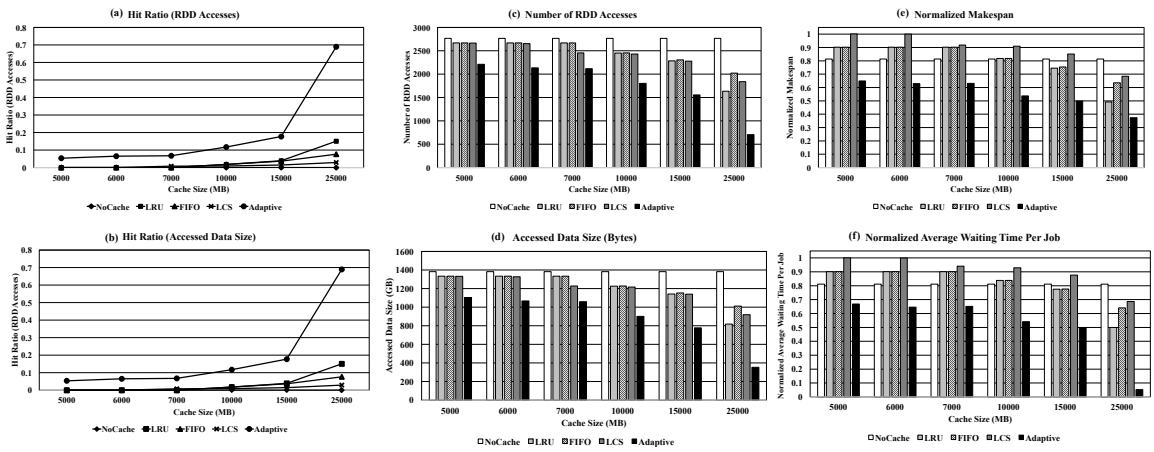



Figure 5.10: Hit ratio, access number and total work makespan results of large scale simulation experiments.

Fig. 5.10 depicts the performance of the five caching algorithms. We conduct a set of simulation experiments by configuring different cache sizes for storing RDDs. Clearly, our algorithm ("Adaptive") significantly improves the hit ratio (up to 70%) across different cache sizes, as seen Figs. 5.10(a) and (b). In contrast, the other algorithms start to hit RDDs (with hit ratio up to 17%) only when the cache capacity becomes large. Consequently, our proposed algorithm reduces the number of RDDs that need to be accessed and calculated (see Figs. 5.10(c) and (d)), which further saves the overall computation costs, i.e., the total work in Figs. 5.10(e) and (f). We also notice that such an improvement from "Adaptive" becomes more significant when we have a larger cache space for RDDs, which indicates that our adaptive algorithm is able to better detect and utilize those shareable and reusable RDDs across jobs.

#### 5.2.5.3 Spark Implementation

We further evaluate our cache algorithm by integrating our methodology into Apache Spark 2.2.1 which is hypervised by VMware Workstation 12.5.0. Table 5.3 summarizes the details of our testbed configuration. In Spark, the memory space is divided into four pools: storage memory, execution memory, unmanaged memory and reserved memory. Only the storage and execution memory pools (i.e., $UnifiedMemoryManager$) are used to store the runtime data of Spark applications. Our implementation focuses on the storage memory, which stores the cached data (RDDs), the internal data propagated through the cluster, and the temporarily unrolled serialized data. Fig. 5.11 further illustrates the main architecture of modules in our implementation. In detail, different from Spark's built-in caching that responds to *persist* and *unpersist* APIs, we build an *RDDCacheManager* module in the *Spark Application Layer* to communicate with cache modules in the *Worker Layer*. Our proposed module maintains statistical records (e.g., historical access, computation overhead, DAG dependency, etc.), and automatically decides which new RDDs to be cached and which existing RDDs to be evicted when the cache space is full.

We select *Ridge Regression* [194] as a benchmark because it is a ubiquitous technique, widely applied in machine learning and data mining applications [195, 196]. The input database we use is a huge table containing thousands of entries (i.e., rows), and each entry has more than ten features (i.e., columns). More than hundred Spark jobs are repeatedly generated with an exponential arrival rate. Each job's DAG contains at least one *Ridge Regression*-related subgraph, which regresses a randomly selected feature column (i.e., target) by a randomly selected subset of the remaining feature columns (i.e., source), i.e., $f_t = \Re(\vec{f_s})$, where $f_t$ is the target feature, and $\Re(\vec{f_s})$ is the

| Component | Specs |
|---|---|
| Host Server | Dell PowerEdge T310 |
| Host Processor | Intel Xeon CPU X3470 |
| Host Processor Speed | 2.93GHz |
| Host Processor Cores | 8 Cores |
| Host Memory Capacity | 16GB DIMM DDR3 |
| Host Memory Data Rate | 1333 MHz |
| Host Hypervisor | VMware Workstation 12.5.0 |
| Big Data Platform | Apache Spark 2.2.1 |
| Storage Device | Western Digital WD20EURS |
| Disk Size | 2TB |
| Disk Bandwidth | SATA 3.0Gbps |
| Memory Size Per Node | 1 GB |
| Disk Size Per Node | 50 GB |

Table 5.3: Testbed configuration.

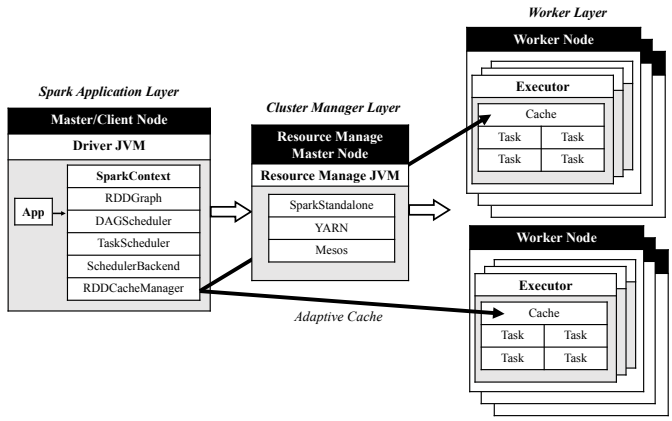



Figure 5.11: Module structure view of our Spark implementation, where our proposed *RDDCacheManager* module cooperates with *cache* module inside each worker node.

regressed correlation function with an input of the source feature vector $\vec{f_s}$. Moreover, different jobs may share the same selections of target and source features, and thus they may have some RDDs with exactly the same generating logic chain (i.e., the subset of DAGs). Unfortunately, the default Spark cannot identify RDDs with the same generating logic chain if they are in *different* jobs. In order to identify these reusable and identical RDDs, our proposed *RDDCacheManager* uses a mapping table to records each RDD's generating logic chain *across* jobs (by importing our customized header files into the benchmark), i.e., we denote $RDD_x$ by a hashing function $key \leftarrow hash(G_x(V, E))$, where $G_x(V, E)$ is the subgraph of $RDD_x$ ($V$ is the set of all ancestor RDDs and $E$ is the set of all operations along the subgraph). Since not all operations are deterministic [197] (e.g., *shuffle* operation on the same input data may result in different RDDs), we only monitor those deterministic operations which guarantee the same output under the same input.

Rather than scrutinizing the cache-friendly case where our adaptive algorithm appears to work well as shown in Sec. 5.2.5.2, it will be more interesting to study the performance under the cache-unfriendly case (also called "stress test" [198]), where the space size of different combinations of source and target features is comparatively large, which causes the production of a large number of different RDDs across jobs. Moreover, the probability of RDDs reaccess is low (e.g., the trace we generated has less than 26% of RDDs are repeated across all jobs), and the temporal distances of RDDs reaccess are also relatively long [62]. Thus, it becomes more challenging for a caching algorithm to make good caching decisions to reduce the total work under such a cache-unfriendly case.

Fig. 5.12 shows the real experimental results under four different caching algorithms, i.e., FIFO, LRU, LCS, and Adaptive. To investigate the impact of cache size, we also change the size of storage memory pool to have different numbers of RDDs that can be cached in that pool. Compared to the other three algorithms, our adaptive algorithm achieves non-negligible improvements on both hit ratio (see in Fig. 5.12(a)) and makespan (see in Fig. 5.12(b)), especially when the cache size increases. Specifically, the hit ratio can be improved by 13% and the makespan can be reduced by 12% at most, which are decent achievements for such a cache-unfriendly stress test with less room to improve performance. Furthermore, we observe that Adaptive significantly increases the hit ratio and reduces the makespan when we have more storage memory space, which again indicates that our caching algorithm has the ability to make good use of memory space. In contrast, the other algorithms have less improvement on the hit ratio and the makespan, since they cannot conduct any cross-job computational overlap detection. While, with a global overview of all accessed RDDs, our adaptive algorithm can effectively select proper RDDs from all jobs to be cached in the limited

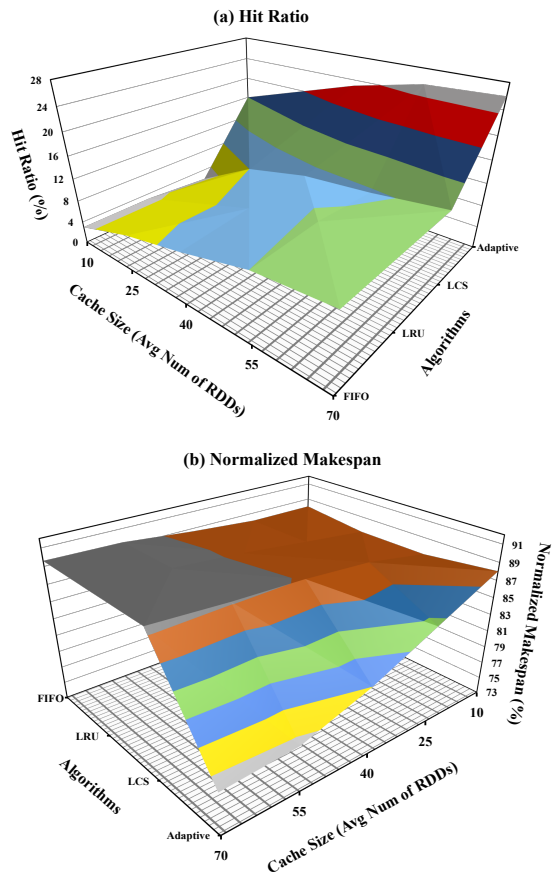



Figure 5.12: Hit ratio and normalized makespan results of a stress testing on cache-unfriendly *Ridge Regression* benchmark with different cache sizes under four cache algorithms.

storage memory pool.

### 5.2.6 Summary

The big data multi-stage parallel computing framework, such as Apache Spark, has been widely used to perform data processing at scale. To speed up the execution, Spark strives to absorb as much intermediate data as possible to the memory to avoid the repeated computation. However, the default in-memory storage mechanism LRU does not choose reasonable RDDs to cache their partitions in the memory, leading to arbitrarily sub-optimal caching decisions. We first formulate the problem by proposing an optimization framework, and then develop an adaptive cache algorithm to store the most valuable intermediate datasets that can be reused in the future. According to our real implementation on Apache Spark, the proposed algorithm can improve the performance by reducing 12% of the total work to re-compute RDDs.

# Chapter 6

# Conclusion and Future Work

## 6.1 Dissertation Conclusion

Cloud computing systems provide shared resources as a pool, which require a central mechanism for resource provisioning and allocation based on user demands. Among different types of demands, we found that the storage I/O is the main bottleneck in cloud computing systems. To address this issue, the industry started to adopt a new type of storage device, Solid State Drives (SSDs), in the server market to improve I/O access performance. Compared to Hard Disk Drives (HDDs), SSDs do not have moving parts. Thus, SSDs can achieve high I/O throughput, low I/O latency and low power consumption. However, SSDs still have the following limitations when being deployed as a tier or a second-level cache or of HDDs: relatively small spatial capacity, high costs-per-GB, limited lifecycle, and write amplification brought by the erase-before-rewrite characteristic. Thus, this dissertation focuses on the flash-based storage resource management in cloud computing datacenter infrastructures, including storage capacity allocation, I/O performance isolation, I/O stack optimization, data replication and recovery, caching and tiering solutions, flash endurance, and total cost of ownership. The dissertation is broken down into three stages: SSD-HDD hybrid datacenter, all-flash datacenter, and big data platform storage optimization.

In order to improve the I/O performance in SSD-HDD hybrid datacenters, we developed a new global SSD resource management scheme called "GREM" to allocate a suitable amount of SSDs to heterogeneous VMs. We showed that GREM can dynamically adjust the partition of SSDs between two zones by leveraging the feedback of workload changes and SSD performance. Meanwhile, aiming to improve the reliability of distributed caching and data processing systems using SSD-HDD hybrid storages, we proposed a data replica manager solution called "AutoReplica".

AutoReplica balances the trade-off between I/O performance and fault tolerance by storing caches in replica nodes' SSDs. In particular, AutoReplica can automatically find the replica nodes with the consideration of network traffic, server geographical distance, and workloads. AutoReplica also supports seamless online migration and parallel prefetching operations.

On the other hand, during the transition from the SSD-HDD hybrid to the all-flash datacenter, we found that simply using existing caching or tiering solutions designed for hybrid storage systems is not enough. Based on our observation that different workloads have different performance improvement after being migrated to a higher-end SSD tier, we developed an automatic data placement manager called "AutoTiering" to associate Virtual Machine Disk Files (VMDKs) with an appropriate SSD tier during runtime. Specifically, AutoTiering considers both historical and predicted performance factors, and the estimated migrating cost. AutoTiering has a micro-benchmark-based sensitivity calibration and regression session to predict virtual machine VM's performance change on different tiers without conducting any actual migration. We evaluated this management scheme on top of the VMware ESXi platform and showed that AutoTiering is able to better utilize the storage resource, optimize the performance, and reduce the migration overhead. Besides, all-flash storages also suffer from the write amplification which further heavily affects the Total Cost of Ownership (TCO). However, there are no standard TCO models available which consider both the write amplification, SSD endurance, and workload characteristics. Thus, we proposed a TCO model to reflect capital and operational costs, the estimated lifetime of SSDs under different workloads, resource restrictions, and Service Level Agreements (SLAs). We also characterized the Write Amplification Factor (WAF) of SSDs as a function of the fraction of sequential writes in a workload, and integrated this function into TCO model. Based on them, we built a novel online workload allocation algorithm MINTCO to dispatch and deploy workloads across SSDs in the storage pool.

After developing performance and reliability enhancement solutions for all-flash datacenters, we investigated how to optimize the I/O stack in the software level of the storage infrastructure of big data platforms. We observed that the major bottleneck in the current deployment of Non-Volatile Memory Express (NVMe) SSDs in the VM-hypervisor environment is caused by multiple non-lock-free intermediate queues. Thus, we presented a hybrid framework, called "H-NVMe", which uses two different modes to fully utilize the NVMe resources (e.g., up to 64K in depth). We showed that H-NVMe's first mode (*Parallel Queue Mode*) can increase parallelism and enable lock-free operations by forwarding jobs from the *Adapter Queue* to our customized enhanced subqueues in the driver. We also showed that H-NVMe's second mode (*Direct Access Mode*) allows trusted applications with VAIO IOFilter [150] attached to their user VMDKs to directly access NVMe

SSDs and bypass the entire I/O stack in the hypervisor layer to further ensure performance isolation. We implement H-NVMe on the VMware ESXi platform, and our evaluation results show that our proposed framework outperforms the original NVMe solution under multiple I/O benchmarks. Apart from hypervisor I/O stack, we also observed that big data multi-stage parallel computing frameworks use very naive in-memory storage mechanisms (e.g., Apache Spark uses LRU) which do not choose reasonable intermediate datasets to cache their partitions in memory, leading to arbitrarily sub-optimal caching decisions. To address this issue, we formulated the problem by proposing an optimization framework, and then developed an adaptive algorithm to cache the most valuable intermediate datasets that can be reused in the future. Our new caching algorithm is able to improve the performance by reducing 12% of the total work time.

## 6.2 Future Work

Storage resource management is still very critical for the big data and cloud computing systems. Thus, in the future, we plan to explore more storage use cases in datacenters and study new storage devices to improve the system efficiency and the overall I/O performance. In particular, we have the following three directions:

**Cost Model and Dispatch Schemes for RAID Mode SSD Arrays:** In the era of flash drivers, there are also lots of available commercial solutions that use RAID mode flash disk arrays in data center to further accelerate the I/O speeds, such as Diff-RAID [199] and RamSan500 [200]. The RAID technique transforms a number of independent disks into a larger and more reliable logical single entity [201]. Motivated by this state-of-the-art trend, we plan to extend MINTCO to support storage systems with RAID mode flash disk arrays. The major problem in this extension is that it is almost impossible to calculate (or even monitor) the exact WAF of RAID mode flash disk arrays during runtime in real implementations. Moreover, for systems with different RAID setups, getting accurate WAF values will be more complicated. Therefore, we plan to develop an approximate approach that aims to estimate both WAF and TCO to guide the datacenter manager to make allocation decisions from a long-term datacenter operation of view.

**Offline Workload-disk Deployment Problem:** Our MINTCO solution is designed for a so-called *online* scenario where the number and the specs of disks in the disk pool are known and fixed. However, the system has no foreknowledge about those I/O workloads that keep coming and waiting to be assigned to the disk pool. In the future, we plan to extend MINTCO to support the *offline* mode scenario, where the datacenter manager needs to allocate *all* known workloads to an

undecided disk pool in the beginning. The major challenge is that besides considering how to allocate workloads to minimize the capital and operating expenditures, our new algorithm should also decide the number and type of disks to run the I/O workloads.

**Investigate New Storage Devices:** Lastly, we plan to investigate new storage devices to further accelerate I/O speed. One promising device is 3D XPoint [30]. Unlike SSD, 3D XPoint does not have the erase-before-rewrite limitation, and theoretically both its I/O speed and lifecycle can be 1000x compared with those of SSD. As mentioned in [202], the arrival of 3D XPoint promises to fundamentally change the memory-storage hierarchy at the hardware, system software, and application levels. We plan to first investigate this new storage device 3D XPoint, and further understand its write amplification characteristics. We also plan to effectively deploy 3D XPoint devices in storage systems by refining our AutoTiering scheme, which will leverage the observation from a recent work [202] that applications such as key-value stores and real-time analytics can benefit the most from 3D XPoint.

# Bibliography


[1] T. N. Theis and H.-S. P. Wong, "The end of moore's law: A new beginning for information technology," *Computing in Science & Engineering*, vol. 19, no. 2, pp. 41–50, 2017.

[2] H. Wang and P. J. Varman, "Balancing fairness and efficiency in tiered storage systems with bottleneck-aware allocation." in *The 12th USENIX Conference on File and Storage Technologies (FAST)*, vol. 14, 2014, pp. 229–242.

[3] "Silicon valley quote," http://siliconvalleyism.com/silicon-valley-quote.php?id=33.

[4] "Big data, for better or worse," https://www.sciencedaily.com/releases/2013/05/130522085217.html.

[5] "2.5 quintillion bytes of data created every day." [Online]. Available: https://goo.gl/K5mGDC

[6] L. A. Barroso, J. Clidaras, and U. Hölzle, "The datacenter as a computer: An introduction to the design of warehouse-scale machines," *Synthesis lectures on computer architecture*, vol. 8, no. 3, pp. 1–154, 2013.

[7] S. John Walker, "Big data: A revolution that will transform how we live, work, and think," 2014.

[8] "RAID," en.wikipedia.org/wiki/RAID.

[9] "Apache Spark," http://spark.apache.org/.

[10] Ssd market history. [Online]. Available: http://www.storagesearch.com/chartingtheriseofssds.html

[11] K. Chodorow, *MongoDB: the definitive guide*. O'Reilly Media, Inc., 2013.

[12] T. Harter, D. Borthakur, S. Dong, A. Aiyer, L. Tang, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Analysis of HDFS under HBase: A Facebook messages case study," in *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST 14)*, 2014, pp. 199–212.

[13] T. Hatanaka, R. Yajima, T. Horiuchi, S. Wang, X. Zhang, M. Takahashi, S. Sakai, and K. Takeuchi, "Ferroelectric (fe)-nand flash memory with non-volatile page buffer for data center application enterprise solid-state drives (ssd)," in *2009 Symposium on VLSI Circuits*. IEEE, 2009, pp. 78–79.

[14] "HP believes all-flash data center has come of age," https://goo.gl/HZULTV.

[15] The all-flash data center transforms business economics & competitiveness. [Online]. Available: https://goo.gl/1GBCYX

[16] X.-Y. Hu, E. Eleftheriou, R. Haas, I. Iliadis, and R. Pletka, "Write amplification analysis in flash-based solid state drives," in *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, 2009, p. 10.

[17] "VMware ESXi," www.vmware.com/products/vsphere-hypervisor.html.

[18] Apache Hadoop. [Online]. Available: http://hadoop.apache.org/

[19] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: cluster computing with working sets," in *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, 2010, pp. 10–10.

[20] Z. Yang, J. Tai, J. Bhimani, J. Wang, N. Mi, and B. Sheng, "GREM: Dynamic SSD Resource Allocation In Virtualized Storage Systems With Heterogeneous IO Workloads," in *35th IEEE International Performance Computing and Communications Conference*. IEEE, 2016.

[21] Z. Yang, J. Wang, D. Evans, and N. Mi, "AutoReplica: Automatic Data Replica Manager in Distributed Caching and Data Processing Systems," in *1st International workshop on Communication, Computing, and Networking in Cyber Physical Systems (CCNCPS)*. IEEE, 2016.

[22] Z. Yang, J. Bhimani, J. Wang, D. Evans, and N. Mi, "Automatic and Scalable Data Replication Manager in Distributed Computation and Storage Infrastructure of Cyber-Physical Systems," in

*Scalable Computing: Practice and Experience, Special Issue on Communication, Computing, and Networking in Cyber-Physical Systems*, 2017.

[23] Z. Yang, M. Hoseinzadeh, A. Andrews, C. Mayers, D. T. Evans, R. T. Bolt, J. Bhimani, N. Mi, and S. Swanson, "AutoTiering: Automatic Data Placement Manager in Multi-Tier All-Flash Datacenter," in *36th IEEE International Performance Computing and Communications Conference*. IEEE, 2017.

[24] Z. Yang, M. Awasthi, M. Ghosh, and N. Mi, "A Fresh Perspective on Total Cost of Ownership Models for Flash Storage in Datacenters," in *2016 IEEE 8th International Conference on Cloud Computing Technology and Science*. IEEE, 2016.

[25] Z. Yang, M. Hoseinzadeh, P. Wong, J. Artoux, C. Mayers, D. T. Evans, R. T. Bolt, J. Bhimani, N. Mi, and S. Swanson, "H-NVMe: A Hybrid Framework of NVMe-based Storage System in Cloud Computing Environment," in *36th IEEE International Performance Computing and Communications Conference (IPCCC)*. IEEE, 2017.

[26] Z. Yang, D. Jia, S. Ioannidis, N. Mi, and B. Sheng, "Intermediate Data Caching Optimization for Multi-Stage and Parallel Big Data Frameworks," in *2018 IEEE International Conference on Cloud Computing (CLOUD 2018)*. IEEE, 2018.

[27] D. I. Shin, Y. J. Yu, and H. Y. Yeom, "Shedding light in the black-box: Structural modeling of modern disk drives," in *Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, 2007. MASCOTS'07. 15th International Symposium on*. IEEE, 2007, pp. 410–417.

[28] H.-S. P. Wong, S. Raoux, S. Kim, J. Liang, J. P. Reifenberg, B. Rajendran, M. Asheghi, and K. E. Goodson, "Phase change memory," *Proceedings of the IEEE*, vol. 98, no. 12, pp. 2201–2227, 2010.

[29] S.-S. Sheu, M.-F. Chang, K.-F. Lin, C.-W. Wu, Y.-S. Chen, P.-F. Chiu, C.-C. Kuo, Y.-S. Yang, P.-C. Chiang, W.-P. Lin *et al.*, "A 4mb embedded slc resistive-ram macro with 7.2 ns read-write random-access time and 160ns mlc-access capability," in *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2011 IEEE International*. IEEE, 2011, pp. 200–202.

[30] I. Newsroom, "Intel and micron produce breakthrough memory technology, july 28, 2015." [Online]. Available: http://newsroom.intel.com/community/intel_newsroom/blog/2015/07/28/intel-and-micron-produce-breakthrough-memory-technology

[31] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee, "Better i/o through byte-addressable, persistent memory," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, 2009, pp. 133–146.

[32] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting phase change memory as a scalable dram alternative," in *ACM SIGARCH Computer Architecture News*, vol. 37, no. 3. ACM, 2009, pp. 2–13.

[33] A. M. Caulfield, J. Coburn, T. Mollov, A. De, A. Akel, J. He, A. Jagatheesan, R. K. Gupta, A. Snavely, and S. Swanson, "Understanding the impact of emerging non-volatile memories on high-performance, io-intensive computing," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society, 2010, pp. 1–11.

[34] H. Kim, S. Seshadri, C. L. Dickey, and L. Chiu, "Evaluating phase change memory for enterprise storage systems: A study of caching and tiering approaches," *ACM Transactions on Storage (TOS)*, vol. 10, no. 4, p. 15, 2014.

[35] E. H. Nam, B. S. J. Kim, H. Eom, and S. L. Min, "Ozone (o3): An out-of-order flash memory controller architecture," *IEEE Transactions on Computers*, vol. 60, no. 5, pp. 653–666, 2011.

[36] "PCI Express Specification," http://pcisig.com/specifications/pciexpress/.

[37] "NVM-EXPRESS," http://www.nvmexpress.org/.

[38] A. Huffman, "NVM Express specification 1.1a," http://www.nvmexpress.org/wp-content/uploads/NVM-Express-1_1a.pdf, Sep 2013.

[39] J. Xu and S. Swanson, "Nova: A log-structured file system for hybrid volatile/non-volatile main memories," in *FAST*, 2016, pp. 323–338.

[40] J. Bhimani, J. Yang, Z. Yang, N. Mi, Q. Xu, M. Awasthi, R. Pandurangan, and V. Balakrishnan, "Understanding Performance of I/O Intensive Containerized Applications for NVMe SSDs," in *35th IEEE International Performance Computing and Communications Conference*. IEEE, 2016.

[41] J. Bhimani, Z. Yang, N. Mi, J. Yang, Q. Xu, M. Awasthi, R. Pandurangan, and V. Balakrishnan, "Docker Container Scheduler for I/O Intensive Applications running on NVMe SSDs," in *IEEE Transactions on Multi-Scale Computing Systems (TMSCS)*, 2018.

[42] A. Awad, B. Kettering, and Y. Solihin, "Non-volatile memory host controller interface performance analysis in high-performance i/o systems," in *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium on*. IEEE, 2015, pp. 145–154.

[43] W. Bux and I. Iliadis, "Performance of greedy garbage collection in flash-based solid-state drives," *Performance Evaluation*, vol. 67, no. 11, pp. 1172–1186, 2010.

[44] P. Desnoyers, "Analytic models of ssd write performance," *ACM Transactions on Storage (TOS)*, vol. 10, no. 2, p. 8, 2014.

[45] R. Agarwal and M. Marrow, "A closed-form expression for write amplification in nand flash," in *GLOBECOM Workshops (GC Wkshps), 2010 IEEE*. IEEE, 2010, pp. 1846–1850.

[46] R. Zhou, M. Liu, and T. Li, "Characterizing the efficiency of data deduplication for big data storage management," in *Workload Characterization (IISWC), 2013 IEEE International Symposium on*. IEEE, 2013, pp. 98–108.

[47] P. Wang, G. Sun, S. Jiang, J. Ouyang, S. Lin, C. Zhang, and J. Cong, "An efficient design and implementation of lsm-tree based key-value store on open-channel ssd," in *Proceedings of the Ninth European Conference on Computer Systems*. ACM, 2014, p. 16.

[48] V. Tarasov, D. Hildebrand, G. Kuenning, and E. Zadok, "Virtual machine workloads: The case for new nas benchmarks," in *Presented as part of the 11th USENIX Conference on File and Storage Technologies (FAST 13)*, 2013, pp. 307–320.

[49] J. Bhimani, Z. Yang, M. Leeser, and N. Mi, "Accelerating Big Data Applications Using Lightweight Virtualization Framework on Enterprise Cloud," in *21st IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2017.

[50] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and linux containers," in *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium On*. IEEE, 2015, pp. 171–172.

[51] N. Megiddo and D. S. Modha, "ARC: A self-tuning, low overhead replacement cache." in *FAST*, vol. 3, 2003, pp. 115–130.

[52] E. O'Neil, P. O'Neil, and G. Weikum, "The LRU-K Page Replacement Algorithm for Database Disk Buffering," in *Proceedings of the 1993 ACM SIGMOD international conference on Management of data*, Washington, DC, 1993, pp. 297–306.

[53] L. A. Belady, "A study of replacement algorithms for a virtual-storage computer," *IBM Systems journal*, vol. 5, no. 2, pp. 78–101, 1966.

[54] S. Bansal and D. S. Modha, "CAR: Clock with adaptive replacement." in *FAST*, vol. 4, 2004, pp. 187–200.

[55] T. Kgil, D. Roberts, and T. Mudge, "Improving NAND Flash Based Disk Caches," in *Proceedings of the 35th Annual International Symposium on Computer Architecture*, Bejing, China, 2008, pp. 327–338.

[56] T. Pritchett and M. Thottethodi, "SieveStore: A Highly-selective, Ensemble-level Disk Cache for Cost-performance," in *Proceedings of the 37th annual international symposium on Computer architecture*, Saint-Malo, France, 2010, pp. 163–174.

[57] F. Chen, D. A. Koufaty, and X. Zhang, "Hystor: Making the Best Use of Solid State Drives in High Performance Storage Systems," in *Proceedings of the International Conference on Supercomputing*, Tucson, Arizona, 2011, pp. 22–32.

[58] M. Wachs, M. Abd-El-Malek, E. Thereska, and G. R. Ganger, "Argon: Performance Insulation for Shared Storage Servers," in *Proceedings of the 5th USENIX conference on File and Storage Technologies*, San Jose, CA, 2007, pp. 61–76.

[59] N. Beckmann and D. Sanchez, "Jigsaw: Scalable Software-Defined Caches," in *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*. IEEE Press, 2013, pp. 213–224.

[60] A. Gulati, A. Merchant, and P. J. Varman, "mclock: handling throughput variability for hypervisor io scheduling," in *Proceedings of the 9th USENIX conference on Operating systems design and implementation*. USENIX Association, 2010, pp. 437–450.

[61] T. Luo, S. Ma, R. Lee, X. Zhang, D. Liu, and L. Zhou, "S-cave: Effective ssd caching to improve virtual machine storage performance," in *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*. IEEE Press, 2013, pp. 103–112.

[62] F. Meng, L. Zhou, X. Ma, S. Uttamchandani, and D. Liu, "vcacheshare: automated server flash cache space management in a virtualization environment," in *USENIX ATC*, 2014.

[63] A. I. Alsalibi, S. Mittal, M. A. Al-Betar, and P. B. Sumari, "A survey of techniques for architecting slc/mlc/tlc hybrid flash memory–based ssds," *Concurrency and Computation: Practice and Experience*, p. e4420, 2018.

[64] D. Gmach, J. Rolia, L. Cherkasova, and A. Kemper, "Resource pool management: Reactive versus proactive or let's be friends," *Computer Networks*, vol. 53, no. 17, pp. 2905–2922, 2009.

[65] T. Setzer and A. Wolke, "Virtual machine re-assignment considering migration overhead," in *Network Operations and Management Symposium (NOMS), 2012 IEEE*. IEEE, 2012, pp. 631–634.

[66] T. K. Petersen and J. Bent, "Hybrid flash arrays for hpc storage systems: An alternative to burst buffers," in *High Performance Extreme Computing Conference (HPEC), 2017 IEEE*. IEEE, 2017, pp. 1–7.

[67] W. Shin, Q. Chen, M. Oh, H. Eom, and H. Y. Yeom, "Os i/o path optimizations for flash solid-state drives." in *USENIX Annual Technical Conference*, 2014, pp. 483–488.

[68] Y. J. Yu, D. I. Shin, W. Shin, N. Y. Song, J. W. Choi, H. S. Kim, H. Eom, and H. Y. Yeom, "Optimizing the block i/o subsystem for fast storage devices," *ACM Transactions on Computer Systems (TOCS)*, vol. 32, no. 2, p. 6, 2014.

[69] H.-J. Kim, Y.-S. Lee, and J.-S. Kim, "Nvmedirect: A user-space i/o framework for application-specific optimization on nvme ssds." in *HotStorage*, 2016.

[70] J. Yang, D. B. Minturn, and F. Hady, "When poll is better than interrupt." in *FAST*, vol. 12, 2012, pp. 3–3.

[71] A. M. Caulfield, T. I. Mollov, L. A. Eisner, A. De, J. Coburn, and S. Swanson, "Providing safe, user space access to fast, solid state disks," *ACM SIGARCH Computer Architecture News*, vol. 40, no. 1, pp. 387–400, 2012.

[72] S. Ahn, K. La, and J. Kim, "Improving i/o resource sharing of linux cgroup for nvme ssds on multi-core systems," in *HotStorage*, 2016.

[73] J. Huang, A. Badam, L. Caulfield, S. Nath, S. Sengupta, B. Sharma, and M. K. Qureshi, "Flashblox: Achieving both performance isolation and uniform lifetime for virtualized ssds." in *FAST*, 2017, pp. 375–390.

[74] J. Bhimani, J. Yang, Z. Yang, N. Mi, N. K. Giri, R. Pandurangan, C. Choi, and V. Balakrishnan, "Enhancing SSDs with Multi-Stream: What? Why? How?" in *36th IEEE International Performance Computing and Communications Conference (IPCCC), Poster Paper*. IEEE, 2017.

[75] J. Bhimani, N. Mi, Z. Yang, J. Yang, R. Pandurangan, C. Choi, and V. Balakrishnan, "FIOS: Feature Based I/O Stream Identification for Improving Endurance of Multi-Stream SSDs," in *2018 IEEE International Conference on Cloud Computing (CLOUD 2018)*. IEEE, 2018.

[76] P. Kumar and H. H. Huang, "Falcon: Scaling io performance in multi-ssd volumes," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. USENIX Association, 2017, pp. 41–53.

[77] Z. Yang, Y. Wang, J. Bhimani, C. C. Tan, and N. Mi, "EAD: Elasticity Aware Deduplication Manager for Datacenters with Multi-tier Storage Systems," in *Cluster Computing*, 2018.

[78] M. Lee, D. H. Kang, M. Lee, and Y. I. Eom, "Improving read performance by isolating multiple queues in nvme ssds," in *Proceedings of the 11th International Conference on Ubiquitous Information Management and Communication*. ACM, 2017, p. 36.

[79] A. M. Caulfield, A. De, J. Coburn, T. I. Mollow, R. K. Gupta, and S. Swanson, "Moneta: A high-performance storage array architecture for next-generation, non-volatile memories," in *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2010, pp. 385–395.

[80] H. Volos, S. Nalli, S. Panneerselvam, V. Varadarajan, P. Saxena, and M. M. Swift, "Aerie: Flexible file-system interfaces to storage-class memory," in *Proceedings of the Ninth European Conference on Computer Systems*. ACM, 2014, p. 14.

[81] A. Verma, R. Koller, L. Useche, and R. Rangaswami, "Srcmap: Energy proportional storage using dynamic consolidation." in *FAST*, vol. 10, 2010, pp. 267–280.

[82] S. Kavalanekar, B. Worthington, Z. Qi, and V. Sharda, "Characterization of Storage Workload Traces from Production Windows Servers," in *Proceedings of the 2008 IEEE International Symposium on Workload Characterization*, Seattle, WA, 2008, pp. 119–128.

[83] R. Koller and R. Rangaswami, "I/o deduplication: Utilizing content similarity to improve i/o performance," *ACM Transactions on Storage (TOS)*, vol. 6, no. 3, p. 13, 2010.

[84] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload analysis of a large-scale key-value store," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 40, no. 1. ACM, 2012, pp. 53–64.

[85] A. Khan, X. Yan, S. Tao, and N. Anerousis, "Workload characterization and prediction in the cloud: A multiple time series approach," in *Network Operations and Management Symposium (NOMS), 2012 IEEE*. IEEE, 2012, pp. 1287–1294.

[86] Z. Yang, H. Fang, Y. Wu, C. Li, B. Zhao, and H. H. Huang, "Understanding the effects of hypervisor i/o scheduling for virtual machine performance interference," in *Cloud Computing Technology and Science (CloudCom), 2012 IEEE 4th International Conference on*. IEEE, 2012, pp. 34–41.

[87] H. Zhang, G. Chen, B. C. Ooi, K.-L. Tan, and M. Zhang, "In-memory big data management and processing: A survey," *IEEE Transactions on Knowledge and Data Engineering*, vol. 27, no. 7, pp. 1920–1948, 2015.

[88] "Apache SINGA," http://singa.incubator.apache.org/en/index.html.

[89] "Apache Giraph," http://giraph.apache.org.

[90] "GridGain," http://www.gridgain.com.

[91] "Apache Storm," http://storm.apache.org.

[92] "Apache Spark Streaming," http://spark.apache.org/streaming.

[93] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears, "Mapreduce online." in *Nsdi*, vol. 10, no. 4, 2010, p. 20.

[94] B. Fitzpatrick, "Distributed Caching with Memcached," *Linux Journal*, vol. 124, no. 5, 2004.

[95] Redis. [Online]. Available: http://redis.io

[96] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh, "Megastore: Providing scalable, highly available storage for interactive services," in *Proceedings of the Conference on Innovative Data system Research (CIDR)*, 2011, pp. 223–234.

[97] L. Xu, M. Li, L. Zhang, A. R. Butt, Y. Wang, and Z. Z. Hu, "Memtune: Dynamic memory management for in-memory data analytic platforms," in *Parallel and Distributed Processing Symposium, 2016 IEEE International*. IEEE, 2016, pp. 383–392.

[98] L. Gu and H. Li, "Memory or time: Performance evaluation for iterative operation on hadoop and spark," in *High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing (HPCC_EUC), 2013 IEEE 10th International Conference on*. IEEE, 2013, pp. 721–727.

[99] T. Wang, J. Wang, N. Nguyen, Z. Yang, N. Mi, and B. Sheng, "EA2S2: An Efficient Application-Aware Storage System for Big Data Processing in Heterogeneous Clusters," in *26th International Conference on Computer Communications and Networks (ICCCN)*. IEEE, 2017.

[100] K. Wang and M. M. H. Khan, "Performance prediction for apache spark platform," in *High Performance Computing and Communications (HPCC), 2015 IEEE 7th International Symposium on Cyberspace Safety and Security (CSS), 2015 IEEE 12th International Conferen on Embedded Software and Systems (ICESS), 2015 IEEE 17th International Conference on*. IEEE, 2015, pp. 166–173.

[101] K. Wang, M. M. H. Khan, N. Nguyen, and S. Gokhale, "Modeling interference for apache spark jobs," in *Cloud Computing (CLOUD), 2016 IEEE 9th International Conference on*. IEEE, 2016, pp. 423–431.

[102] Y. Geng, X. Shi, C. Pei, H. Jin, and W. Jiang, "Lcs: an efficient data eviction strategy for spark," *International Journal of Parallel Programming*, vol. 45, no. 6, pp. 1285–1297, 2017.

[103] Y. Yu, W. Wang, J. Zhang, and K. B. Letaief, "Lrc: Dependency-aware cache management for data analytics clusters," in *Conference on Computer Communications*. IEEE, 2017, pp. 1–9.

[104] M. Duan, K. Li, Z. Tang, G. Xiao, and K. Li, "Selection and replacement algorithms for memory performance improvement in spark," *Concurrency and Computation: Practice and Experience*, vol. 28, no. 8, pp. 2473–2486, 2016.

[105] K. Wang, K. Zhang, and C. Gao, "A new scheme for cache optimization based on cluster computing framework spark," in *Computational Intelligence and Design (ISCID), 2015 8th International Symposium on*, vol. 1. IEEE, 2015, pp. 114–117.

[106] A. Kathpal, M. Kulkarni, and A. Bakre, "Analyzing compute vs. storage tradeoff for video-aware storage efficiency." in *HotStorage*, 2012.

[107] S. Ponnusamy and E. Karthikeyan, "Cache optimization on hot-point proxy caching using weighted-rank cache replacement policy," *ETRI journal*, vol. 35, no. 4, pp. 687–696, 2013.

[108] I. Iliadis, E. K. Kolodner, D. Sotnikov, P. K. Ta-Shma, and V. Venkatesan, "Enhancing reliability of a storage system by strategic replica placement and migration," Apr. 25 2017, uS Patent 9,635,109.

[109] B. Chen, Z. Yang, S. Huang, X. Du, Z. Cui, J. Bhimani, and N. Mi, "Cyber-Physical System Enabled Nearby Traffic Flow Modelling for Autonomous Vehicles," in *36th IEEE International Performance Computing and Communications Conference, Special Session on Cyber Physical Systems: Security, Computing, and Performance (IPCCC-CPS)*. IEEE, 2017.

[110] A. Cidon, S. Rumble, R. Stutsman, S. Katti, J. Ousterhout, and M. Rosenblum, "Copysets: Reducing the frequency of data loss in cloud storage," in *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*, 2013, pp. 37–48.

[111] J. Zhang, G. Wu, X. Hu, and X. Wu, "A distributed cache for hadoop distributed file system in real-time cloud services," in *2012 ACM/IEEE 13th International Conference on Grid Computing*. IEEE, 2012, pp. 12–21.

[112] J. Wang, T. Wang, Z. Yang, N. Mi, and S. Bo, "eSplash: Efficient Speculation in Large Scale Heterogeneous Computing Systems," in *35th IEEE International Performance Computing and Communications Conference*. IEEE, 2016.

[113] Z. Yang, J. Bhimani, Y. Yao, C.-H. Lin, J. Wang, N. Mi, and B. Sheng, "AutoAdmin: Admission Control in YARN Clusters Based on Dynamic Resource Reservation," in *Scalable*

*Computing: Practice and Experience, Special Issue on Advances in Emerging Wireless Communications and Networking*, vol. 19, 2018, pp. 53–67.

[114] J. Wang, T. Wang, Z. Yang, Y. Mao, N. Mi, and B. Sheng, "SEINA: A Stealthy and Effective Internal Attack in Hadoop Systems," in *International Conference on Computing, Networking and Communications (ICNC 2017)*. IEEE, 2017.

[115] H. Gao, Z. Yang, J. Bhimani, T. Wang, J. Wang, B. Sheng, and N. Mi, "AutoPath: Harnessing Parallel Execution Paths for Efficient Resource Allocation in Multi-Stage Big Data Frameworks," in *26th International Conference on Computer Communications and Networks (ICCCN)*. IEEE, 2017.

[116] Y. Zhao, C. Li, L. Li, and P. Zhang, "Dynamic replica creation strategy based on file heat and node load in hybrid cloud," in *Advanced Communication Technology (ICACT), 2017 19th International Conference on*. IEEE, 2017, pp. 213–220.

[117] R. K. Grace and R. Manimegalai, "Dynamic replica placement and selection strategies in data grids—a comprehensive survey," *Journal of Parallel and Distributed Computing*, vol. 74, no. 2, pp. 2099–2108, 2014.

[118] R. Li, W. Feng, H. Wu, and Q. Huang, "A replication strategy for a distributed high-speed caching system based on spatiotemporal access patterns of geospatial data," *Computers, Environment and Urban Systems*, 2014.

[119] P. L. Suresh, M. Canini, S. Schmid, and A. Feldmann, "C3: Cutting Tail Latency in Cloud Data Stores via Adaptive Replica Selection," in *NSDI*, 2015, pp. 513–527.

[120] G. L. Wade and J. M. Haile, "Data replica control," Jan. 31 2017, uS Patent 9,558,074.

[121] R. E. Wagner, "Automated method for identifying and repairing logical data discrepancies between database replicas in a database cluster," Feb. 28 2012, uS Patent 8,126,848.

[122] N. S. Islam, X. Lu, M. Wasi-ur Rahman, D. Shankar, and D. K. Panda, "Triple-H: A hybrid approach to accelerate HDFS on HPC clusters with heterogeneous storage architecture," in *Cluster, Cloud and Grid Computing (CCGrid), 2015 15th IEEE/ACM International Symposium on*. IEEE, 2015, pp. 101–110.

[123] L. Gkatzikis, V. Sourlas, C. Fischione, I. Koutsopoulos, and G. Dán, "Clustered content replication for hierarchical content delivery networks," in *2015 IEEE International Conference on Communications (ICC)*. IEEE, 2015, pp. 5872–5877.

[124] A. Floratou and J. M. Patel, "Replica placement in multi-tenant database environments," in *2015 IEEE International Congress on Big Data*. IEEE, 2015, pp. 246–253.

[125] J. Guerra, H. Pucha, J. S. Glider, W. Belluomini, and R. Rangaswami, "Cost effective storage using extent based dynamic tiering." in *FAST*, 2011, pp. 273–286.

[126] D. Narayanan, E. Thereska, A. Donnelly, S. Elnikety, and A. Rowstron, "Migrating server storage to ssds: Analysis of tradeoffs," in *Proceedings of the 4th ACM European conference on Computer systems*, Nuremberg, Germany, 2009, pp. 145–158.

[127] V. Mohan, S. Sankar, S. Gurumurthi, and W. Redmond, "refresh ssds: Enabling high endurance, low cost flash in datacenters," *Univ. of Virginia, Tech. Rep. CS-2012-05*, 2012.

[128] Z. Li, A. Mukker, and E. Zadok, "On the importance of evaluating storage systems' costs," in *Proceedings of the 6th USENIX Conference on Hot Topics in Storage and File Systems (HotStorage '14)*, 2014, pp. 1–5.

[129] Y. Cheng, M. S. Iqbal, A. Gupta, A. R. Butt, and V. Tech, "Pricing games for hybrid object stores in the cloud: provider vs. tenant," in *Proceedings of the 7th USENIX Conference on Hot Topics in Cloud Computing*. USENIX Association, 2015, pp. 20–20.

[130] S. Ghandeharizadeh, S. Irani, and J. Lam, "Memory hierarchy design for caching middleware in the age of nvm," 2015.

[131] Y. Cui, C. Ingalz, T. Gao, and A. Heydari, "Total cost of ownership model for data center technology evaluation," in *Thermal and Thermomechanical Phenomena in Electronic Systems (ITherm), 2017 16th IEEE Intersociety Conference on*. IEEE, 2017, pp. 936–942.

[132] B. Jun and D. Shin, "Workload-aware budget compensation scheduling for nvme solid state drives," in *Non-Volatile Memory System and Applications Symposium (NVMSA), 2015 IEEE*. IEEE, 2015, pp. 1–6.

[133] D. Narayanan, A. Donnelly, and A. Rowstron, "Write off-loading: Practical power management for enterprise storage," *ACM Transactions on Storage*, vol. 4, no. 3, pp. 10:1–10:23, 2008.

[134] "UMass Trace Repository," http://traces.cs.umass.edu/index.php/Storage/Storage.

[135] C. Ding and Y. Zhong, "Predicting whole-program locality through reuse distance analysis," in *ACM SIGPLAN Notices*, vol. 38, no. 5. ACM, 2003, pp. 245–257.

[136] "Dell fluid cache for storage area networks," http://www.dell.com/learn/us/en/04/solutions/fluid-cache-san.

[137] S. Byan, J. Lentini, A. Madan, L. Pabon, M. Condict, J. Kimmel, S. Kleiman, C. Small, and M. Storer, "Mercury: Host-side flash caching for the data center," in *IEEE 28th Symposium on Mass Storage Systems and Technologies*, Pacific Grove, CA, 2012, pp. 1–12.

[138] H. Kim, I. Koltsidas, N. Ioannou, S. Seshadri, P. Muench, C. L. Dickey, and L. Chiu, "Flash-conscious cache population for enterprise database workloads." in *ADMS@ VLDB*, 2014, pp. 45–56.

[139] D. Liu, N. Mi, J. Tai, X. Zhu, and J. Lo, "Vfrm: Flash resource manager in vmware esx server," in *Network Operations and Management Symposium (NOMS), 2014 IEEE*. IEEE, 2014, pp. 1–7.

[140] J. Tai, D. Liu, Z. Yang, X. Zhu, J. Lo, and N. Mi, "Improving Flash Resource Utilization at Minimal Management Cost in Virtualized Flash-based Storage Systems," *Cloud Computing, IEEE Transactions on*, July 2017.

[141] S. M. Tonni, M. Z. Rahman, S. Parvin, and A. Gawanmeh, "Securing big data efficiently through microaggregation technique," in *Distributed Computing Systems Workshops (ICDCSW), 2017 IEEE 37th International Conference on*. IEEE, 2017, pp. 125–130.

[142] J. Roemer, M. Groman, Z. Yang, Y. Wang, C. C. Tan, and N. Mi, "Improving Virtual Machine Migration via Deduplication," in *11th IEEE International Conference on Mobile Ad Hoc and Sensor Systems (MASS 2014)*. IEEE, 2014, pp. 702–707.

[143] J. Bhimani, N. Mi, M. Leeser, and Z. Yang, "FiM: Performance Prediction Model for Parallel Computation in Iterative Data Processing Applications," in *10th IEEE International Conference on Cloud Computing (CLOUD)*. IEEE, 2017.

[144] "RAID Penalty," http://theithollow.com/2012/03/21/understanding-raid-penalty/.

[145] "dstat," https://dag.wiee.rs/home-made/dstat.

[146] "iostat," https://linux.die.net/man/1/iostat.

[147] "blktrace," https://linux.die.net/man/8/blktrace.

[148] J. Shi, J. Wan, H. Yan, and H. Suo, "A survey of cyber-physical systems," in *Wireless Communications and Signal Processing (WCSP), 2011 International Conference on*. IEEE, 2011, pp. 1–6.

[149] M. S. Hossain, "Cloud-supported cyber–physical localization framework for patients monitoring," *IEEE Systems Journal*, vol. 11, no. 1, pp. 118–127, 2017.

[150] "vsphere apis for i/o filtering (vaio) program," https://code.vmware.com/programs/vsphere-apis-for-io-filtering.

[151] "Intel IOMeter," http://www.iometer.org.

[152] "FIO: Flexible I/O Tester," http://linux.die.net/man/1/fio.

[153] L. Wang, J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi, S. Zhang *et al.*, "Bigdatabench: A big data benchmark suite from internet services," in *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*. IEEE, 2014, pp. 488–499.

[154] "Amazon Web Services," http://aws.amazon.com/.

[155] "Google Computer Engine," http://cloud.google.com/compute/.

[156] "The mega-datacenter battle update: Internet giants increased capex in 2014," http://451research.com/report-short?entityId=84745.

[157] P. Desnoyers, "Analytic modeling of ssd write performance," in *Proceedings of the 5th Annual International Systems and Storage Conference*. ACM, 2012, p. 12.

[158] "MSR Cambridge Traces," http://iotta.snia.org/traces/388.

[159] "SNIA Iotta Repository," http://iotta.snia.org/historical_section.

[160] M. Oh, H. Eom, and H. Y. Yeom, "Enhancing the i/o system for virtual machines using high performance ssds," in *Performance Computing and Communications Conference (IPCCC), 2014 IEEE International*. IEEE, 2014, pp. 1–8.

[161] "VMware Pluggable Storage Architecture Package." [Online]. Available: https://code.vmware.com/vmware-ready-programs/storage/psa

[162] "SCSI Command." [Online]. Available: https://en.wikipedia.org/wiki/SCSI_command

[163] "Performance Best Practices for VMware vSphere 6.5, VMware ESXi 6.5, vCenter Server 6.5," http://www.vmware.com/support/2017.

[164] "MMC Queue." [Online]. Available: https://en.wikipedia.org/wiki/M/M/c_queue

[165] "Erlang C Formula." [Online]. Available: https://en.wikipedia.org/wiki/Erlang_(unit)#Erlang_C_formula

[166] O. F. Koch and A. Benlian, "The effect of free sampling strategies on freemium conversion rates," *Electronic Markets*, vol. 27, no. 1, pp. 67–76, 2017.

[167] "Context switching."

[168] D. Maier, *Theory of relational databases*. Computer Science Pr, 1983.

[169] T. Hastie, R. Tibshirani, and J. Friedman, "The elements of statistical learning: data mining, inference and prediction," *New York: Springer-Verlag*, vol. 1, no. 8, pp. 371–406, 2001.

[170] Y. C. Eldar and G. Kutyniok, *Compressed sensing: theory and applications*. Cambridge University Press, 2012.

[171] S. A. Jyothi, C. Curino, I. Menache, S. M. Narayanamurthy, A. Tumanov, J. Yaniv, R. Mavlyutov, I. Goiri, S. Krishnan, J. Kulkarni *et al.*, "Morpheus: Towards automated slos for enterprise clusters." in *OSDI*, 2016, pp. 117–134.

[172] Y. Chen, S. Alspaugh, and R. Katz, "Interactive analytical processing in big data systems: A cross-industry study of mapreduce workloads," *Proceedings of the VLDB Endowment*, vol. 5, no. 12, pp. 1802–1813, 2012.

[173] S. Ioannidis and E. Yeh, "Adaptive caching networks with optimality guarantees," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 44, no. 1. ACM, 2016, pp. 113–124.

[174] K. Shanmugam, N. Golrezaei, A. G. Dimakis, A. F. Molisch, and G. Caire, "Femtocaching: Wireless content delivery through distributed caching helpers," *Transactions on Information Theory*, vol. 59, no. 12, pp. 8402–8413, 2013.

[175] J. Dean, S. Ghemawat, and G. Inc, "Mapreduce: simplified data processing on large clusters," in *OSDI'04*, 2004.

[176] M. Zaharia, M. Chowdhury, T. Das *et al.*, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *9th USENIX Symposium on Networked Systems Design and Implementation*, 2012.

[177] R. Barrett, "Under the covers of the google app engine datastore," *Google I/O, San Francisco, CA*, 2008.

[178] "Apache HBase," http://hbase.apache.org/.

[179] B. Scholkopf and A. J. Smola, *Learning with kernels: support vector machines, regularization, optimization, and beyond*. MIT press, 2001.

[180] I. Jolliffe, *Principal component analysis*. Wiley Online Library, 2002.

[181] K. Gregor, I. Danihelka, A. Mnih, C. Blundell, and D. Wierstra, "Deep autoregressive networks," in *Proceedings of The 31st International Conference on Machine Learning*, 2014, pp. 1242–1250.

[182] L. Fleischer, M. X. Goemans, V. S. Mirrokni, and M. Sviridenko, "Tight approximation algorithms for maximum separable assignment problems," *Mathematics of Operations Research*, vol. 36, no. 3, pp. 416–431, 2011.

[183] N. Megiddo and D. Modha, "ARC: A Self-Tuning, Low Overhead Replacement Cache," in *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, San Francisco, CA, 2003, pp. 115–130.

[184] D. Lee, J. Choi, J.-H. Kim, S. Noh, S. L. Min, Y. Cho, and C. S. Kim, "LRFU: A Spectrum of Policies that Subsumes the Least Recently Used and Least Frequently Used Policies," *IEEE Transactions on Computers*, vol. 50, no. 12, pp. 1352–1361, 2001.

[185] G. Nemhauser, L. Wolsey, and M. Fisher, "An analysis of approximations for maximizing submodular set functions," *Mathematical Programming*, vol. 14, no. 1, pp. 265–294, 1978.

[186] M. Sviridenko, "A note on maximizing a submodular set function subject to a knapsack constraint," *Oper. Res. Lett.*, vol. 32, no. 1, pp. 41–43, 2004.

[187] A. Krause and C. Guestrin, "A note on the budgeted maximization of submodular functions," CMU, Tech. Rep.

[188] A. Kulik, H. Shachnai, and T. Tamir, "Maximizing submodular set functions subject to multiple linear constraints," in *Proceedings of the twentieth annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics, 2009, pp. 545–554.

[189] A. A. Ageev and M. I. Sviridenko, "Pipage rounding: A new method of constructing algorithms with proven performance guarantee," *Journal of Combinatorial Optimization*, vol. 8, no. 3, pp. 307–328, 2004.

[190] C. Chekuri, J. Vondrak, and R. Zenklusen, "Dependent randomized rounding via exchange properties of combinatorial structures," in *FOCS*. IEEE Computer Society, 2010.

[191] "Ewma chart," http://en.wikipedia.org/wiki/EWMA_chart.

[192] A. Nemirovski, *Efficient methods in convex programming*, 2005.

[193] T. Horel, S. Ioannidis, and S. Muthukrishnan, "Budget feasible mechanisms for experimental design," in *Latin American Symposium on Theoretical Informatics*. Springer, 2014, pp. 719–730.

[194] A. E. Hoerl and R. W. Kennard, "Ridge regression: Biased estimation for nonorthogonal problems," *Technometrics*, vol. 12, no. 1, pp. 55–67, 1970.

[195] G. Li and P. Niu, "An enhanced extreme learning machine based on ridge regression for regression," *Neural Computing and Applications*, vol. 22, no. 3-4, pp. 803–810, 2013.

[196] G.-B. Huang, H. Zhou, X. Ding, and R. Zhang, "Extreme learning machine for regression and multiclass classification," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 42, no. 2, pp. 513–529, 2012.

[197] "Spark Programming Guide," http://spark.apache.org/docs/1.6.2/programming-guide.html.

[198] I. Wagner, V. Bertacco, and T. Austin, "Stresstest: an automatic approach to test generation via activity monitors," in *Proceedings of the 42nd annual Design Automation Conference*. ACM, 2005, pp. 783–788.

[199] M. Balakrishnan, A. Kadav, V. Prabhakaran, and D. Malkhi, "Differential raid: Rethinking raid for ssd reliability," *ACM Transactions on Storage (TOS)*, vol. 6, no. 2, p. 4, 2010.

[200] W. Hutsell, "An in-depth look at the ramsan-500 cached flash solid state disk," *Texas Memory Systems White Paper*, p. 16, 2008.

[201] D. Narayanan, E. Thereska, A. Donnelly, S. Elnikety, and A. Rowstron, "Migrating server storage to ssds: analysis of tradeoffs," in *Proceedings of the 4th ACM European conference on Computer systems*. ACM, 2009, pp. 145–158.

[202] F. T. Hady, A. Foong, B. Veal, and D. Williams, "Platform storage performance with 3d xpoint technology," *Proceedings of the IEEE*, vol. 105, no. 9, pp. 1822–1833, 2017.