The Multi-tiered Future of Storage: Understanding Cost and Performance Trade-offs in Modern Storage Systems

Muhammad Safdar Iqbal

Keywords: Multi-tier storage, persistent memory, cloud storage, in-memory cache, multi-level cache, MapReduce, analytics workflows, pricing games, dynamic pricing.

The Multi-tiered Future of Storage: Understanding Cost and Performance Trade-offs in Modern Storage Systems

Muhammad Safdar Iqbal

(ABSTRACT)

In the last decade, the landscape of storage hardware and software has changed considerably. Storage hardware has diversified from hard disk drives and solid state drives to include persistent memory (PMEM) devices such as phase change memory (PCM) and Flash-backed DRAM. On the software side, the increasing adoption of cloud services for building and deploying consumer and enterprise applications is driving the use of cloud storage services. Cloud providers have responded by providing a plethora of choices of storage services, each of which have unique performance characteristics and pricing. We argue this variety represents an opportunity for modern storage systems, and it can be leveraged to improve operational costs of the systems.

We propose that storage tiering is an effective technique for balancing operational or deployment costs and performance in such modern storage systems. We demonstrate this via three key techniques. First, thmCache, which leverages tiering to conserve the lifetime of PMEM devices, hence saving hardware upgrade costs. Second, CAST, which leverages tiering between multiple types of cloud storage to deliver higher utility (i.e. performance per unit of cost) for cloud tenants. Third, we propose a dynamic pricing scheme for cloud storage services, which leverages tiering to increase the cloud provider's profit or offset their management costs.

This work was sponsored in part by the NSF under Grants CNS-1615411, CNS-1565314, CCF-0746832, CNS-1405697 and CNS-142278.

The Multi-tiered Future of Storage: Understanding Cost and Performance Trade-offs in Modern Storage Systems

Muhammad Safdar Iqbal

(GENERAL AUDIENCE ABSTRACT)

Storage and retrival of data is one of the key functions of any computer system. Improvements in hardware and software related to data storage can help computer users store (a) store the data faster, which makes for overall faster performance; and (b) increase the storage capacity, which helps store the increasing amount of data generated by modern computer users. Typically, most computers are equipped with either a hard disk drive (HDD) or, the newer and faster, solid state drive (SSD) for data storage. In the last decade however, the landscape of data storage hardware and software has advanced considerably. On the hardware side, several hardware makers are introducing *persistent memory* (PMEM) devices, which provide very high speed, high capacity storage at reasonable price points. On the software side, the increasing adoption of cloud services by software developers that are building and operating consumer and enterprise applications is driving the use of *cloud storage services*. These services allow the developers to store a large amount of data without having to manage any physical hardware, paying for the service on a usage-based pricing structure. However, every application's speed and capacity needs are not the same; hence, cloud service providers have responded by providing a plethora of choices of storage services, each of which have unique performance characteristics and pricing. We argue this variety represents an opportunity for modern storage systems, and it can be leveraged to improve the operating costs of the systems.

Storage tiering is a classical technique that involves partitioning the stored data and placing each partition in a different storage device. This lets the applications use mulitple devices at once, taking advantage of each's sterngths and mitigating their weaknesses. We propose that storage tiering is a relevant and effective technique for balancing operational or deployment costs and performance in modern storage systems such as PMEM devices and cloud storage services. We demonstrate this via three key techniques. First, THMCACHE, which leverages tiering between multiple types of storage hardware to conserve the lifetime of PMEM devices, hence saving hardware upgrade costs. Second, CAST, which leverages tiering between multiple types of cloud storage services to deliver higher utility (i.e. performance per unit of cost) for software developers using these services. Third, we propose a dynamic pricing scheme for cloud storage services, which leverages tiering between multiple cloud storage services to increase the cloud service provider's profit or offset their management costs.

Dedication

Dedicated to my family
for their endless love, encouragement and support.

Acknowledgments

I owe my gratitude to a great many people who have contributed toward making this thesis possible. I am deeply indebted to my advisor Dr. Ali R. Butt for his trust, guidance and support during my time as a graduate student at Virginia Tech. He has always found time in his busy schedule for countless discussions and feedback sessions. He has never clipped my wings and has always given me the freedom to explore new ideas. Beyond being just my research advisor, there have been times when he has also donned the hat of a compassionate mentor, helping me up through some tough times both in my personal and professional life. His counsel has been instrumental in helping me recover whenever my steps have faltered. I would like to personally thank the members of my dissertation committee: Dr. Dennis Kafura and Dr. Eli Tilevich for their valuable comments and suggestions. It is indeed a great honor for me to have them as a part of my committee. I would like to thank Dr. Kirk Cameron for his support and advice during my time at VT. I would also like to thank Dr. Jin Baek Kwon from Sun Moon University (South Korea), Sandeep Shete from NetApp/Primary Data and Dr. Qasim Ali from VMware for guiding my research direction.

The Distributed Systems and Storage Lab (DSSL) at VT has been the perfect ecosystem for my professional and personal development. I am indebted to the senior colleagues and friends in the lab, Yue, Krish, Hyogi, Bo, Hafeez and Henri for their consistent support and guidance throughout my time in DSSL. I am also thankful to my peers Ali Anwar, Luna, Bharti, Arnab, Salman, Jamal, Uday, Jin and Sangeetha for my making DSSL fun, memorable and productive. It has been an honor to work alongside each and every one of them. I would like to thank the staff of the VT Computer Science department for their support, particularly the technical support staff (Rob, Matt and Ryan) and the administrative staff (Debbie, Sharon, Megan, Emily and Melanie).

I am thankful to all my friends at VT, Taha, Ali, Asad, Burhan and Affan for their friendship and support. Finally, and most importantly, the love and patience of my family has been and will continue to remain the backbone of all my endeavors. My parents, Yousaf and Zahida; my brothers, Haider and Mubasher have showered heir unconditional love and unwavering support all throughout my life. I would also like to express my heartfelt gratitude to my undergraduate advisors at LUMS, Dr. Zartash Uzmi and Dr. Ihsan Qazi, whose guidance and support went a long way in motivating and preparing me for graduate school.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In the last decade, the landscape of storage hardware and software has changed considerably. On the hardware side, hard disk drives (HDDs) and DRAM devices have been supplanted by fast, reliable solid-state drives (SSDs) [31]. Moreover, *persistent memory* devices are making their way into the market [22, 24]. These new hardware devices have challenged the assumptions inherent in the design of traditional storage systems, presented a new set of challenges such as asynchronous read/write latencies and relatively low endurance [26]. On the software sides, there are two major trends: (1) the rise of cloud-based storage services, which serve both as active storage for cloud-deployed applications [13, 10] and as archival storage [3]; and (2) new application workloads such as large key-value/NoSQL stores [102, 74] and data analytics application frameworks [6, 59]. Figure 1.1 shows the multitude of options available to system designers and application developers both in the local storage hardware and cloud storage services. With the improvement in network connectivity and emergence of new data sources such as Internet of Things (IoT) endpoints, mobile platforms [8, 5], and wearable devices, [15, 18], enterprise-scale data-intensive analytics now involves terabyte- to petabyte-scale data with more data being generated from these sources constantly. Thus, storage allocation and management would play a key role in overall performance improvement and cost reduction for this domain.

Tiered storage has been used in many contexts throughout the history of storage system design. There has been lots of work to balance the HDD–SSD cost and benefits by distributing the workload on a hybrid medium consisting of multiple tiers [56, 65, 73, 94], each tier comprised of HDDs, SSDs or another type storage devices. For instance, techniques such as hot-cold storage tiering [65, 68] and caching [57, 32] are being employed to reduce the I/O cost by storing, replicating, or pre-staging data with different access frequencies in multi-tiered storage mediums.

In this thesis, we argue that **storage tiering is an effective technique for balancing operational or deployment costs and performance in modern storage systems**. We develop thmCache for persistent memory devices to show how ideas from storage

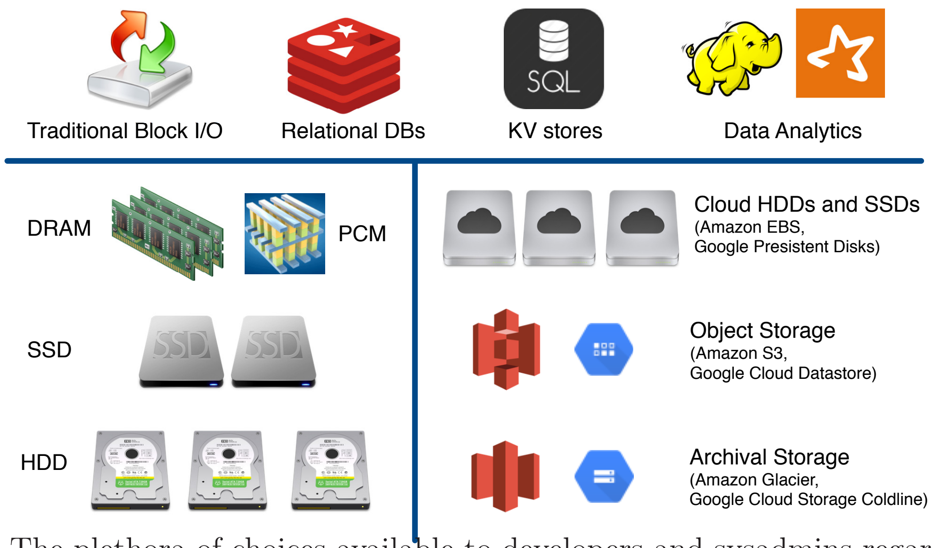


Figure 1.1: The plethora of choices available to developers and sysadmins regarding storage hardware and cloud storage services.

tiering and multi-level caching can be combined to conserve the lifetime of devices with low write-endurance. We design CAST, a storage provisioning system for cloud storage services, that shows how workload-aware tiering can significantly reduce storage costs for cloud tenants when deploying data analytics applications in the cloud. Finally, through trace-driven simulations we show that intelligent pricing techniques combined with tiered storage can be used by cloud providers to improve their operating profits. Hence, we show how storage tiering is effective in taking advantage of the variety offered by modern storage systems.

1.1 Persistent Memory and thmCache

Persistent memory (PMEM) technologies are beginning to be considered for use as main memory in the systems of near future. Desirable properties such as persistence, byte-addressability, and low access latency are driving such uses for PMEM [50, 98]. PMEM technologies such as PCM also offer high density, which is a key property for supporting modern applications that generate and ingest a large amount of data and have stringent latency requirements. However, the main drawback of PCM is its low write-endurance, which makes it challenging to sustain traditional DRAM workloads on PCM devices while maintaining a reasonable device lifetime [61, 103].

To help storage application developers alleviate this problem, we designed thmCache[1], a 'tiered' cache design that enables combining multiple persistent memory devices to conserve the lifetime of a low endurance device such as PCM using a high endurance device such as

[1]thmCache is named after the beloved Two-Headed Monster [28] from Seasme Street.

DRAM on top. thmCache keeps write-intensive cache blocks in the DRAM tier to absorb the writes, thereby reducing the write traffic to the PCM tier. We designed our solution as block-based cache that can be used as a disk cache (similar to Flashcache[16, 64]) or as file system buffer cache [37].

We implement our design in the AccuSim [40] trace-driven cache simulator and use industry standard workload traces for evaluation. Our evaluation shows that compared to a non-tiered PCM-only cache with the ARC scheme, thmCache reduces the number of PCM writes in write-intensive workloads by 75% on average, correspondingly improving device lifetime, and uses DRAM that is only 10–15% of the PCM size, while maintaining a hit ratio that is more than 99% that of the non-tiered configuration.

1.2 Cloud Storage Provisioning and Pricing

The cloud computing paradigm provides powerful computation capabilities, high scalability and resource elasticity at reduced operational and administration costs. The use of cloud resources frees tenants from the traditionally cumbersome IT infrastructure planning and maintenance, and allows them to focus on application development and optimal resource deployment. These desirable features coupled with the advances in virtualization infrastructure are driving the adoption of public, private, and hybrid clouds for not only web applications, such as Netflix, Instagram and Airbnb, but also modern big data analytics using parallel programming paradigms such as Hadoop [6] and Dryad [59].

Enterprises are increasingly moving their big data analytics to the cloud with the goal of reducing costs without sacrificing application performance [9, 23] Cloud service providers offer their tenants a myriad of storage options, which while flexible, makes the choice of storage deployment non trivial. Crafting deployment scenarios to leverage these choices in a cost-effective manner — under the unique pricing models and multi-tenancy dynamics of the cloud environment — presents unique challenges in designing cloud-based data analytics frameworks.

1.2.1 Workload-aware Provisioning: CAST

CAST, a Cloud Analytics Storage Tiering solution, enables cloud tenants can use to reduce monetary cost and improve performance of analytics workloads. The approach takes the first step towards providing storage tiering support for data analytics in the cloud. CAST performs offline workload profiling to construct job performance prediction models on different cloud storage services, and combines these models with workload specifications and high-level tenant goals to generate a cost-effective data placement and storage provisioning plan. Furthermore, we build CAST++ to enhance CAST's optimization model by incorporating data reuse patterns and across-jobs interdependencies common in realistic analytics workloads.

Tests with production workload traces from Facebook and a 400-core Google Cloud based Hadoop cluster demonstrate that CAST++ achieves 1.21× performance and reduces deployment costs by 51.4% compared to local storage configuration. Finally, this work has appeared in a competitive, peer-reviewed conference [45] in 2015.

1.2.2 Dynamic Pricing Games

Cloud object stores, such as Amazon S3 [4], Google Cloud Storage [17] and Azure Storage [10], are versatile storage services that support a wide range of use cases. They simplify the management of large blocks of data at scale and hence are becoming the *de facto* storage choice for big data analytics platforms. To build these services cost-effectively, cloud providers use hard disk drives (HDDs) in their object store deployments. However, the lower performance of HDDs affect tenants who have strict performance requirements for their big data applications. The use of faster storage devices such as solid state drives (SSDs) is thus desirable by the tenants, but incurs significant maintenance costs to the provider.

To help the cloud providers offset maintenance costs of SSDs, we propose a game-theoretic approach to cloud storage pricing. We design a tiered object store for the cloud, which comprises both fast and slow storage devices. The resulting hybrid store exposes the tiering to tenants with a dynamic pricing model that is based on the tenants' usage and the provider's desire to maximize profits. The tenants leverage knowledge of their workloads and current pricing information to select a data placement strategy that would meet the application requirements at the lowest cost. Providers, in turn, monitor tenant usage of all tiers, and adjust their prices to extract more profit from the usage of expensive tiers. Our approach allows both a service provider and its tenants to engage in a pricing game, which our results show yields a win–win situation.

This work appeared in a competitive, peer-reviewed workshop [46] and was subsequently invited to appear in a peer-reviewed magazine for a wider technical audience [47].

1.3 Related Work

Before we delve into the details of all three works in the subsequent chapters, we provide a brief background of classical storage tiering research. Most of the work on tiering focuses on ***hot/cold data classification-based tiering***, which involves periodically monitoring the access frequencies of files or blocks in a storage system and using these frequencies as the basis for data movement between tiers. For example, recent research [65, 73, 94] has focused on improving storage cost and utilization efficiency by placing hot/cold data in different storage tiers. Guerra et. al.[56] builds an SSD-based dynamic tiering system to minimize cost and power consumption, and existing works handle file system and block level I/Os (e.g., 4 – 32 KB) for POSIX-style workloads (e.g., server, database, file systems, etc.). However, the cost model and tiering mechanism used in prior approaches cannot be directly applied to in-memory caching systems for PMEM or analytics batch processing applications running in a public cloud environment. This is mainly due to workload heterogeneity in in-memory caches and cloud storage and analytics. In contrast, our work provides insights into design of a tiered storage management framework for a wide variety of local storage and cloud-based workloads.

We also highlight techniques for enterprise data management that explore ideas beyond tiering. As enterprises are increasingly moving their infrastructure services to the cloud with the goal of reducing the administration cost, advanced techniques should be used to enhance the quality of service experience of data-intensive applications and reduce the cost for both cloud service providers and tenants [44, 43, 42, 33, 34, 35, 36].

Chapter 2

thmCache: Multi-tier Caching for Persistent Memory Devices

In the last decade, persistent memory (PMEM) technologies such as phase change memory (PCM) [61, 103] have been explored as a complement to or a replacement for DRAM-based main memory. The primary benefits of PMEM include persistence, byte-addressability and access latency within two to four times that of DRAM. PMEM are also high density, which is an important property given the recent rise in the use of in-memory computing [102, 74], for example, in-memory big data frameworks for real-time data analytics and in-memory key-value stores for scalable cloud services. These services are very demanding in terms of capacity. Big data frameworks have to support increasingly large datasets and in-memory key-value stores have to serve large user bases that grow as the applications they serve become popular. The scalability challenges of conventional DRAM technology limit its ability to meet this increasing demand. Therefore, PMEM technologies can provide a viable alternative to DRAM for these use cases.

PMEM devices are being realized via various different technologies such as traditional Flash-backed DRAM [62], phase change memory (PCM) [61, 103], memristors [93] and spin torque transfer magnetic RAM (STT-MRAM) [92]. Each technology exhibits its own set of performance and scalability characteristics. For example, Flash-backed DRAM has the read-write latency of traditional DRAM and its write endurance is also very similar to that of DRAM. However, this technology suffers from DRAM's scalability challenges [38]. PCM, on the other hand, has higher read and write latencies and lower endurance than DRAM. Table 2.1 shows a comparison of the characteristics of different PMEM technologies gathered from data available about state-of-the-art prototypes and products in recent literature [98, 38].

Supporting traditional in-memory workloads in PMEM devices with such varied characteristics is a challenge because of two reasons: (a) the low write endurance of PCM; and (b) the increased (and asymmetric) read/write latency of PCM as compared to STT-MRAM or Flash-backed DRAM. The extant approach for overcoming these challenges primarily in-

Table 2.1: Characteristics of different types of memory technologies.

Device Type	Read Latency (ns)	Write Latency (ns)	Endurance (# of writes per bit)
SRAM	2–3	2–3	∞
DRAM	7.5–15	7.5–15	10^{15}–10^{18}
PCM	48–70	150–220	10^{8}–10^{12}
STT-MRAM	5–30	10–100	$\sim 10^{15}$

volves coupling a small amount of say, Flash-backed DRAM, which has high endurance and low latency, with PCM, which has low-endurance and high latency. Some proposals include changes to the memory hierarchy, where the Flash-backed DRAM is used as an embedded cache in the PCM device and is not managed as a separate PMEM device by the OS [87]. This approach requires minimal changes in OS subsystems such as the virtual memory manager and the disk buffer cache. However, there is no opportunity for the OS or the application to perform any optimization based on specific requirements or a knowledge of the workloads. Another approach is to make both PCM and Flash-backed DRAM visible to the OS, which manages both of them as part of a single address space [48, 50]. Following the conventions set forth in existing work, we call this architecture *hybrid persistent memory*. This architecture gives more choices to system designers as the OS can either expose both high-endurance and low-endurance PMEM to application developers or manage the two transparently to overcome the endurance differential.

In this paper, we present THMCACHE, a multi-tiered cache design that aims to conserve the lifetime of low-endurance PMEM devices (such as PCM) by reducing their write traffic. We focus on the hybrid memory architecture and group the different PMEM devices into two *tiers*: a high-endurance (HE) tier with Flash-backed DRAM or STT-MRAM, and a low-endurance (LE) tier with PCM. The architecture of THMCACHE is inspired by works such as multi-level exclusive caching with multi-level collaboration [54]. We use a write-only cache replacement scheme in the high-endurance tier because it has been shown that write references are a better predictor of future writes, than both read and write references [71]. By using the HE tier for absorbing write operations to write-hot blocks, THMCACHE aims to reduce the write traffic to the LE tier and hence conserve its lifetime.

This chapter details the following research contributions: (1) We present the design of THMCACHE that exploits the Adaptive Replacement Cache (ARC) scheme [81]. Each tier uses the ARC replacement scheme to track and evict blocks. The HE tier ARC only tracks writes while the LE tier ARC tracks both reads and writes. (2) We implement THMCACHE in a simulator based on AccuSim [40, 1], a cache simulator with support for multiple cache replacement schemes. (3) We evaluate THMCACHE using two sets of traces: MSR Cambridge traces [25, 83] and traces we collected from the Filebench [14, 80] benchmark suite. Our experiments show that compared to a non-tiered cache (that uses ARC), THMCACHE reduces the number of writes in the LE tier by as high as 75%, correspondingly improving device lifetime. To achieve this 75% reduction, thmCache needs a HE tier size that is only 15% of the LE tier size. This reduction is achieved while maintaining a hit ratio that is 99% that of the non-tiered configuration. Even for Filebench workloads with higher read-to-write ratios, thmCache reduces the number of LE tier writes by up to 30%, while needing an HE tier that is only 10–15% of the LE tier size and maintaining a hit ratio that is more than 95% that of the non-tiered configuration.

2.1 Background and Related Work

Extending the lifetime of low-endurance persistent memory devices is an area of active research. There have been proposals to design persistent memory-aware cache replacement schemes as well as approaches to conserve lifetime and use persistent memory in storage system design.

2.1.1 Persistent Memory-Aware Cache Replacement

Partially safe buffers [30] is an early work on designing a buffer cache for the hybrid memory architecture. The focus of this work is on designing replacement schemes that exploit the persistent nature of the devices and preferably perform writes in the persistent (or *safe*) region of the memory. However, this technique is not suitable for low-endurance PMEM devices such as PCM, as the technique directs most writes towards the persistent region. CLOCK-DWF [71], M-CLOCK [70], and LDF-CLOCK [99] also target increasing the lifespan of PCM devices by reducing their write traffic. However, the schemes specifically target page replacement in the OS virtual memory manager and only consider lightweight approaches derived from the CLOCK replacement scheme.

H-ARC [52] assumes a storage hierarchy with persistent memory devices such as PCM as the sole main memory with Flash-based SSD as the backing store, instead of the hybrid memory architecture targeted in our work. The goal of H-ARC is to reduce write traffic to Flash by reducing the frequency of dirty page synchronization from the buffer cache. H-ARC divides the cache into a clean region and a dirty region at the first level, and a recency region and a frequency region at the second level. The sizes of the regions on both levels are adapted using a scheme similar to ARC. The CFLRU [85] and LRU-WSR [63] schemes also optimize for reducing the number of writebacks to Flash. In contrast, our design is concerned with reducing writes to the low-endurance tier inside a multi-tiered cache.

2.1.2 General Approaches for Multi-level Caching

The state-of-the-art work on caching for the hybrid memory architecture is inspired by earlier work on multi-level caches. MQ [104] is a cache replacement scheme for second-level caches, which uses multiple LRU queues for maintaining blocks of different access frequency for different periods of time. Multi-level exclusive caching techniques such as Demote [96] and Promote [54] propose a framework for designing multi-level versions of existing replacement schemes such as LRU and ARC. These schemes focus on multi-level cache hierarchies with individual caches on different network hosts, and prioritize improving the caches' hit ratios and overall access latency of the operations. More recently, there has been some work on *non-datapath caches*, caches that are not required to make a cache update on every cache miss. LARC [58] and mARC [88] propose schemes that employ opportunistic caching, i.e., they skip cache updates in order to achieve a better hit ratio. Our technique bypasses the high-endurance memory tier for read updates, which is an example of opportunistic caching; however our focus is on reducing write traffic to the low-endurance memory tier.

2.1.3 Alternate System Designs for Persistent Memory

Qureshi et. al. [87] and Ferriera et. al. [53] propose hybrid memory architectures that improve PCM write latency and lifetime using multiple hardware-based techniques such as (a) fine-grained writeback to remove unnessary writes from a page, (b) even distribution of writes (i.e. wear-leveling) in PCM via periodic remaping of virtual pages to different physical pages, and (c) write buffering techniques for the DRAM. These hardware based techniques propose changes to memory controller and the memory management units and may prove difficult to integrate into existing systems. In addition, fine-grained writeback and wear-leveling techniques are complimentary to the techniques presented in this paper.

There have also been proposals to exploit different properties of persistent memory technologies to design novel storage architectures. UBJ [69] is a buffer cache architecture that exploits the persistence property of PMEM devices to fuse journaling and buffer cache and reduce write amplification caused by journaling file systems. Echo [39] is a key value store that targets providing fine-grained transactions and consistency for in-memory key value stores designed for persistent memory devices. BPFS [48] and PMFS [50] are file systems designed specifically for persistent memory that forgo the block-oriented nature of disk devices and redesign file system operations for byte-addressability. These techniques represent a clean-slate approach to designing storage software for PMEM devices and are complimentary to our work.

2.2 Design

We design thmCache with the goal of reducing the write traffic to the low-endurance (LE) PCM device in a hybrid main memory architecture consisting of PCM and high-endurance (HE) Flash-backed DRAM. In the hybrid memory architecture, both the LE and the HE tiers are visible to the OS as separate regions in a single physical address space. The thmCache design makes use of both types of devices, as follows: the HE tier acts as the highest level cache, while the the LE tier acts as a lower level cache. Similarly as in other tiered storage systems, there are two key choices in our design:

(a) *What cache blocks should be migrated to the HE tier?* The purpose of the HE tier is to cache frequently written pages, while the LE tier caters to the rest of the working set. Therefore, we track pages that are *write-hot*, i.e., have a write frequency higher than the rest of the cache blocks that are still resident in memory. The replacement policy in the HE tier only takes write operations into account, demoting any *write-cold* blocks to the LE tier.

(b) *What scheme should we use for determining write-hot data and migrating it to the HE tier?* In thmCache, we use a two-stage approach to determining write-hot cache blocks. When a block is first inserted into the cache, it resides in the LE tier. In addition to managing the rest of the working set, the LE tier maintains a list of recently-written blocks that are candidates to be promoted to the HE tier. When the block receives its first write operation, it becomes a candidate for migration to the HE tier. On receiving the second write operation, the block is migrated to the HE tier, and the operation is completed in the HE tier. Once the block is there, the HE tier manages it using a cache replacement scheme that only tracks write operations.

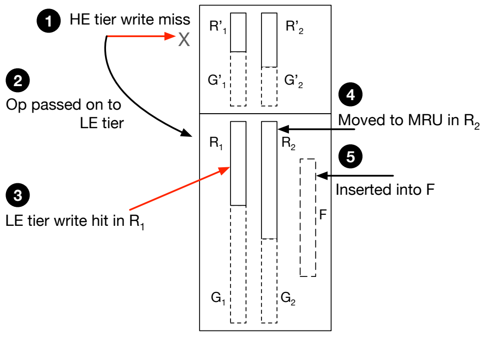
Note that the two tiers of thmCache use slightly different cache replacement policies. This helps each tier to attune their replacement policies via tracking hot and cold pages as needed, and evicting pages when the cache is full.

2.2.1 Adaptive Replacement Cache (ARC)

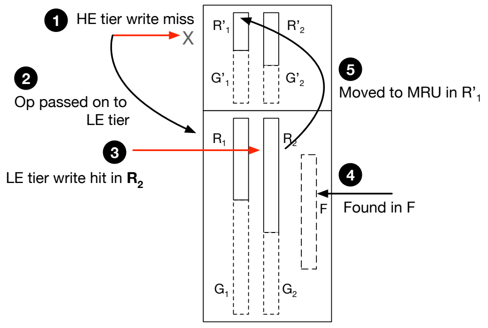
In this paper, we use Adaptive Replacement Cache (ARC) [81] for both the LE tier and HE tier. Adaptive Replacement Cache (ARC) is a cache replacement scheme, which under unpredictable workload, tries to decide the proper cache allocation for recently used pages and frequently used pages. The basic idea of ARC is to partition the cache into two queues, each managed using LRU: the first queue contains blocks accessed only once, while the second contains blocks accessed more than once. A hit to the first queue moves the accessed block to the second queue. Moreover, ARC maintains a ghost queue for each one of the 'real' queues. Whenever a block is evicted from the cache, its metadata is retained in the ghost cache. A hit to a block whose history information is retained in the ghost cache also causes the block to move to the second queue and enables ARC to adapt the size of the corresponding real queue. This allows ARC to increase its 'investment' in the queue that can provide a better hit ratio. Similar to LRU, ARC has a constant complexity per request.

In THMCACHE, the LE tier uses ARC to track both read and write operations. However, the HE tier only tracks write operations. If a read operation results in a hit in the HE tier, even though the block is served from the HE tier, its position in the HE ARC queues is not changed. However, on a write operation, a hit results in moving the block to the MRU position in the second queue. We focus on writes exclusively in order to prioritize the HE tier for `write-hot` blocks. This is motivated by previous work [71] that shows that write history is a better indicator of future writes than a combined read and write history.

2.2.2 THMCACHE Scheme



(a) Block selected as candidate for promotion.



(b) Block promoted to DRAM.

Figure 2.1: THMCACHE block promotion from the LE tier to the HE tier. R and R' denote real queues, B and B' denote ghost queues, F denotes FIFO queue. Note that F contains only the metadata of these blocks; the actual blocks are on R_2.

Figure 2.1 shows the THMCACHE scheme for tracking write-hot pages. We first describe the actions that are taken whenever there is a hit or miss in either one of the tiers. Note that we denote the real and ghost queues as R and G, respectively. For LE tier, real queues are R_1 (accessed-once) and R_2 (accessed-at-least-twice), and the corresponding ghost queues as G_1 and G_2. Similarly, the HE queues are denoted by R'_1, R'_2, G'_1, and G'_2.

Hit in the HE tier

When there is a hit in HE, the current operation is performed in the HE tier. If this current operation is a write, the block is moved to the MRU position in R'_2 in accordance with the ARC scheme. Otherwise, the block retains its current position in HE real queues R'_1 or R'_2.

Miss in the HE tier

If there is a miss in HE, we check if the block, i.e., its metadata, is in HE ghost queues G'_1 or G'_2. One of the two cases then arises:

- If the block is resident in the ghost queues, and the operation is a write, the HE tier asks the LE tier to promote the block. The write operation is then performed in HE. Otherwise, if the operation is a read, it is performed directly in the LE tier. Note that the adaptation of queue sizes on a ghost hit is also done as per the ARC scheme, but only if the operation is a write.
- If the block is not found in the ghost queues, the HE tier simply passes the operation to the LE tier. However, if the LE tier independently decides to promote the block, the operation is still performed in the HE tier and the block is moved to the MRU position in R'_1.

Hit in the LE tier

When there is a miss in the HE tier, the current operation is passed on to the LE tier [Steps 1 and 2 in Figures 2.1(a) and 2.1(b)].

If there is a hit in LE, the block is moved to the MRU position in LE real queue R_2 [Steps 3 and 4 in Figure 2.1(a)]. If the operation is a write, we check if the block's metadata is present in the **filter queue** F. The filter queue F acts as a sliding window for $|F|$ recently written blocks. It filters out any blocks that are written only once, such as parts of a write-only scan. This helps prevent any existing write-hot blocks from being unnecessarily evicted from the HE tier. If the block is not found in F, it is inserted [Step 5 in 2.1(a)].

If the block is found in F [Steps 3 and 4 in Figure 2.1(b)], it is promoted to the HE tier and the operation is then completed there [Step 5 in Figure 2.1(b)]. This block is placed in R'_1, as since being in the HE tier, it was written only once. Note that if a block is removed from the LE tier (either as a promotion or eviction), it is also removed from F.

Miss in the LE tier

If there is a miss in LE, the block is read from the disk and placed at the MRU position in R_1. Also, if the block is present in any of the LE ghost queues, queue size adaption in accordance with the ARC scheme is also performed, irrespective of if the operation is a read or a write.

Eviction

When a block is promoted to the HE tier when it is full, the block to be evicted to make the space is selected from either R_1' or R_2' by the ARC scheme, depending on which queue does it want to 'invest' more. To accommodate this block, the LE tier might also evict a block, which is similarly selected from either R_1 or R_2.

The overall goal of thmCache is to use the HE tier for tracking *write-hot* blocks and evict *write-cold* blocks to the LE tier. It also tries to keep out blocks that are part of write-only scans such as data dumps. Note that all the metadata required for managing the two instances of ARC (both the LE and HE) is kept in the *non-persistent* DRAM as the metadata is write-intensive given it is updated on every operation.

2.3 Evaluation

This section presents an evaluation of thmCache via a trace-driven simulation.

2.3.1 Experimental Setup

We implemented thmCache in AccuSim [40, 1]. AccuSim is a trace driven simulator for I/O workloads, originally built to faithfully simulate the Linux buffer cache. We updated AccuSim with two new features: (a) support for the hybrid main memory architecture and an implementation of our ARC-based thmCache scheme, and (b) support for the trace format used by the MSR Cambridge traces [25].

In our experiments, we assume phase change memory (PCM) as our low-endurance PMEM tier. For the high-endurance tier, we assume the classical, volatile DRAM as a proxy for state-of-the-art PMEM devices such as STT-MRAM or Flash-backed DRAM. Hence, we use the terms "LE tier" and "PCM tier" interchangeably in this section. Similarly, we interchange "HE tier" and "DRAM tier" in this section. To show how much lifetime we are able to conserve, we compare thmCache against a non-tiered cache using the vanilla ARC scheme. The non-tired ARC cache in each experiment has a size equal to the total memory (PCM + DRAM) available to the corresponding tiered cache.

2.3.2 Workloads

We used the following two sets of workload traces in our evaluation:

MSR Cambridge Traces

The MSR Cambridge traces were released by Narayanan et. al. [83] and made available through SNIA [25]. These are block-level traces collected from enterprise servers, and categorized based on the server workload. For example, the `prn` traces are collected from a print server and `proj` traces from a file server supporting project directories. Please note that each server contains multiple volumes and the traces for each volume are provided separately; however, since traces from the same server have similar patterns, we use the traces from the first volume of each server. Table 2.2 describes the relevant characteristics of these traces. All the selected traces are write-intensive.

Table 2.2: Characteristics of workloads

Workload	Purpose	Working Set Size (KB)	Total Accesses	Read-Write Ratio
`proj_0`	Project directories	3, 332, 448	40, 366, 612	1 : 2.03
`hm_0`	Hardware monitoring	2, 441, 616	8, 985, 487	1 : 2.06
`prn_0`	Print server	15, 546, 188	17, 635, 766	1 : 4.05
`fileserver`	Network file server	5, 132, 224	84, 479, 573	1.58 : 1
`oltp`	OLTP	2, 368, 148	10, 748, 211	3.84 : 1

Filebench Traces

The second set of traces used in our evaluation are synthetic traces generated using the Filebench I/O benchmark [14] suite. We used Filebench to generate synthetic workloads using two popular workload models `fileserver` and `oltp`. Each workload model was configured so as to yield a representative trace of the modeled application. Table 2.2 also shows the details about these traces including working set size and ratio of read/write operations. The Filebench traces have much higher reads-to-writes ratios than the MSR Cambridge traces described in Section 2.3.2.

The detailed traces of the workloads were obtained by modifying the *strace* [27] Linux utility. *Strace* intercepts the system calls of the traced process and was modified to record the following information about the I/O operations: access type, time, file identifier (`inode`), starting block offset and I/O size in blocks.

2.3.3 PCM Write Traffic

In our evaluation, we primarily try to answer the following question: *How much PCM write endurance can be conserved using a certain amount of DRAM?* To answer this question we observe the PCM write traffic under thmCache with different DRAM sizes (expressed as percentage of the PCM size). The lesser the write traffic, the more writes THMCACHE conserves, hence improving the lifetime of the PCM device. Note that PCM writes are measured as the number of times cache blocks were written to the PCM during the course of the workload; this includes admissions to the PCM tier, writes performed on a block while it resides in the PCM tier, and the evictions from the DRAM tier.

The number of PCM writes incurred by THMCACHE are normalized by the number of PCM writes incurred in *non-tiered PCM-only cache* managed using the ARC replacement scheme. This helps to show the proportion of writes that are conserved by THMCACHE. For this experiment, we set the PCM size to 25%, 50% and 70% of the working set size of the each trace trace. This allows us to emulate the projected large capacity for PCM devices [71]. We vary the size of DRAM as 5%, 10%, 15% and 20% of the PCM size and measure the number of write operations performed on the PCM device for each size.

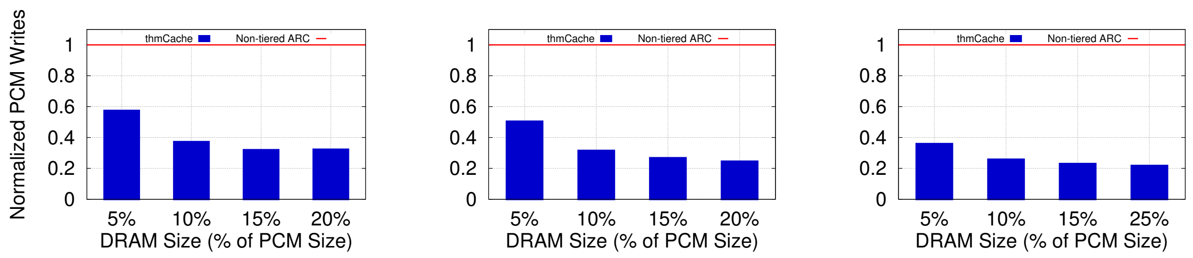
MSR Cambridge Traces

Figures 2.2, 2.3 and 2.4 show the results of this experiment for MSR Cambridge traces. There are two key takeaways from these results: (a) MSR Cambridge traces used in these experiments are write-intensive and also show a high degree of reuse. Therefore, THMCACHE is able to greatly reduce the write traffic to PCM. For example, when PCM is 50% of working set size, for the `hm_0` and `prn_0` workloads, using a DRAM size only 5% of the PCM size reduces the PCM write traffic by $50-60\%$. When using a DRAM size that is $10-15\%$ of the PCM size, this reduction in PCM write traffic increases to $68-75\%$. The `proj_0` workload experiences a sharp drop in PCM write traffic when the DRAM size is increased from 5% to 10% of PCM size. This is because the majority of frequently written blocks in this workload cannot fit in the smaller DRAM size and therefore they have to be evicted to PCM. But as the DRAM size becomes sufficient, the PCM write traffic drops considerably. (b) As the PCM size increases, the amount of DRAM required to achieve a certain percentage reduction in writes remains a manageable size. For example, for the `hm_0` workload, a reduction of 68% can be achieved by using 20% of DRAM when using a PCM 25% that of the working set size. When the PCM size is increased to 50% of the working set size, the DRAM required to achieve the same reduction in PCM write traffic is 10%.

Filebench Traces

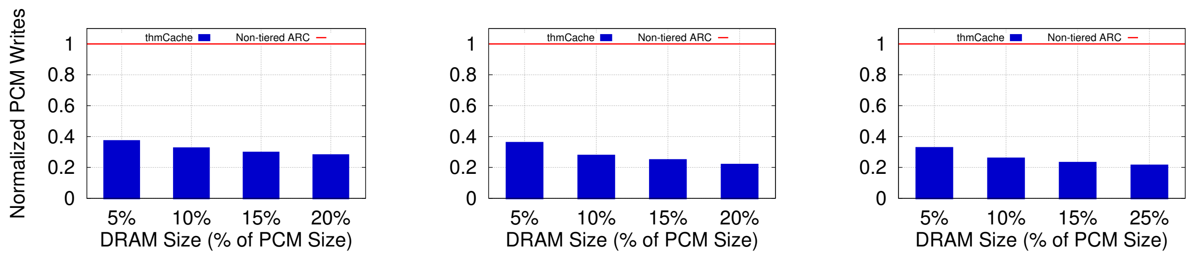
Figure 2.5 shows the number of PCM writes incurred by THMCACHE when running the `fileserver` workload trace from Filebench. Figure 2.6 shows the results for the `oltp` workload trace.

For the filebench traces, the reduction in PCM writes is not as high as for the MSR Cambridge traces. For example, in Figure 2.5(b) using DRAM equal to 10% and 20% of PCM size yields a reduction of 22% and 43% in writes respectively. Similarly, in Figure 2.5(c), we observe



(a) PCM Size = 25% of Working Set Size (b) PCM Size = 50% of Working Set Size (c) PCM Size = 75% of Working Set Size

Figure 2.2: Normalized PCM writes incurred by THMCACHE for `hm_0` MSR Cambridge trace.



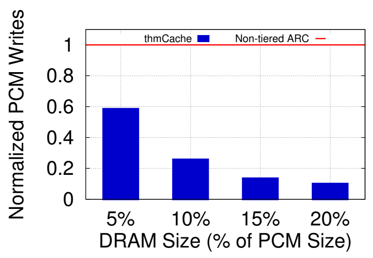
(a) PCM Size = 25% of Working Set Size (b) PCM Size = 50% of Working Set Size (c) PCM Size = 75% of Working Set Size

Figure 2.3: Normalized PCM writes incurred by THMCACHE for `prn_0` MSR Cambridge trace.

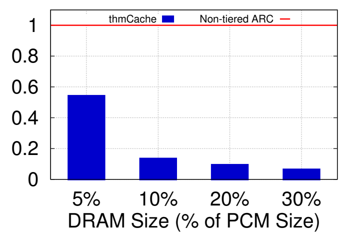
that using 10% of PCM size as the DRAM size yields a reduction of 29% in PCM writes. In Figure 2.5(a), where the size of PCM is 50% of the working set size, we observe that using DRAM capacity equal to 5% of PCM capacity yields a reduction of 8% in the PCM write traffic whereas using 20% yields a reduction of about 30%. `oltp` shows a similar trend. The maximum reduction in PCM writes achieved in Figure 2.6(a) (75% of working set size) is 19% whereas that achieved in Figure 2.6(b) (100% of working set size) is 37%. This is because these workload traces have a larger read request volume and write operations are less frequent, so it becomes harder for THMCACHE to track a set of frequently written blocks to be kept in the DRAM tier. Please note, however, that when the sizes of PCM are 75% and 100% of the working set size, we see that percentage reduction in write traffic becomes more than 2× that of the percentage of PCM size used as the DRAM size.

2.3.4 Hit Ratio

In this section, we evaluate the overall hit ratio of THMCACHE. For these experiments, we set the PCM size to be 25%, 50%, and 75%. We use the same four DRAM sizes (5%, 10%, 15% and 20% of the PCM size used in the experiment) as used in Section 2.3.3. The

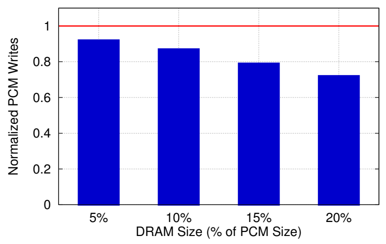


(a) PCM Size = 50% of Working Set Size

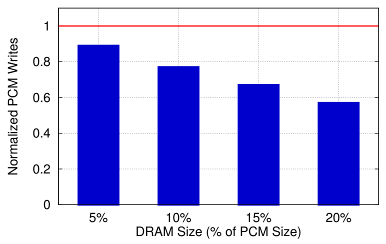


(b) PCM Size = 75% of Working Set Size

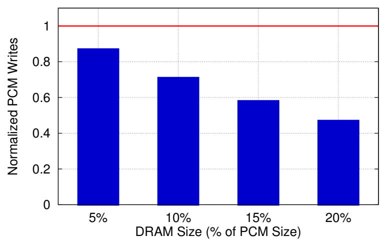
Figure 2.4: Normalized PCM writes incurred by thmCache for `proj_0` MSR Cambridge trace.



(a) PCM Size = 50% of Working Set Size



(b) PCM Size = 75% of Working Set Size



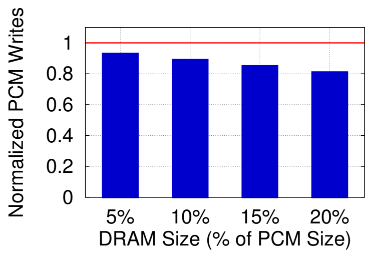
(c) PCM Size = 100% of Working Set Size

Figure 2.5: PCM writes incurred by thmCache for `fileserver`

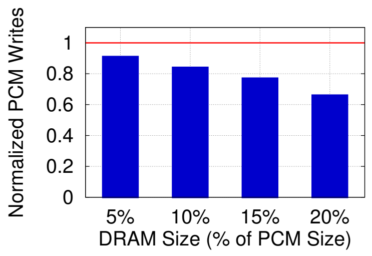
cache size used for each non-tiered configuration is equal to the total size (PCM + DRAM) of matching tiered configuration. We average the hit ratio over the four DRAM size to calculate an average hit ratio for each PCM size used, under both thmCache and non-tiered ARC. We then compare the hit ratio achieved for each PCM size with its corresponding PCM-only non-tiered cache managed by ARC to show the effect of thmCache on hit ratio.

Figure 2.7 shows the result of this experiment for the MSR Cambridge traces `hm_0`, `prn_0` and `proj_0`. For every trace, the hit ratio observed under thmCache is very close or equal to that observed under non-tiered ARC. *The difference between the two hit ratios is less than or equal to* 0.01 *for all the three workloads.* Figure 2.8 shows the results for the `fileserver` trace. Here the difference between the hit ratios of thmCache and non-tiered ARC ranges from 0.04 for 25% PCM to only 0.01 for 75% PCM. We observe that in all cases the hit ratio of thmCache is more than 95% that of the non-tiered PCM-only ARC. Hence, even in the case of read-intensive workloads, the overall hit ratio of thmCache is not affected significantly.

This shows that the thmCache design can achieve the reduction in PCM writes with a negligible impact on the hit ratio of the cache.

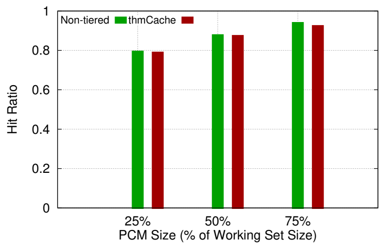


(a) PCM Size = 75% of Working Set Size

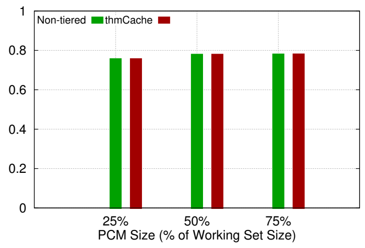


(b) PCM Size = 100% of Working Set Size

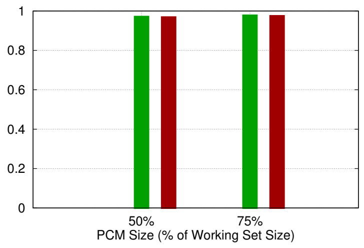
Figure 2.6: PCM writes incurred by THMCACHE for `oltp`



(a) hm_0



(b) prn_0



(c) proj_0

Figure 2.7: Hit ratio of THMCACHE vs that of non-tiered ARC for MSR Cambridge traces.

2.4 Summary

In this paper, we presented THMCACHE, a tiered cache design that conserves the lifetime of low-endurance persistent memory devices such as PCM by combining them with high-endurance PMEM devices. THMCACHE achieves this by caching write-intensive blocks in the high-endurance tier in order to absorb as many writes as possible. This way, our approach reduces the write traffic to the low-endurance tier. We extended the AccuSim cache simulator to support a hybrid PMEM architecture and implemented THMCACHE in it. Our evaluation

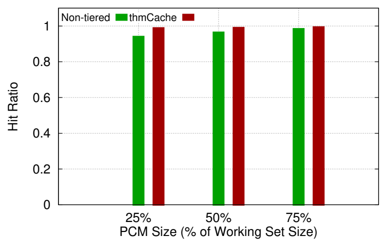


Figure 2.8: Hit ratio of THMCACHE vs that of non-tiered ARC for `fileserver`

shows that for write-intensive workloads, thmCache can achieve on average 75% reduction in PCM write traffic using DRAM of size 15% that of the PCM size, while maintaining a hit ratio that is over 99% of that achieved using a non-tiered, PCM-only, ARC-based cache. Thus, our approach offers a practical solution for realizing tiered caching systems for overcoming the drawbacks of persistent memory devices.

Chapter 3

CAST: Tiering Storage for Data Analytics in the Cloud

With the growth of cloud-based platforms and services, cloud-based data analytics, using parallel programming paradigms such as Hadoop [6] and Dryad [59], have found their way into public clouds. Cloud providers such as Amazon Web Services, Google Cloud, and Microsoft Azure, have started providing data analytics platform as a service [2, 19, 23], which is being adopted widely.

While cloud makes data analytics easy to deploy and scale, the vast variety of available storage services with different persistence, performance and capacity characteristics, presents unique challenges for deploying big data analytics in the cloud. For example, Google Cloud Platform provides four different storage options as listed in Table 3.1. While `ephSSD` offers the highest sequential and random I/O performance, it does not provide data persistence (data stored in `ephSSD` is lost once the associated VMs are terminated). Network-attached persistent block storage services using `persHDD` or `persSSD` as storage media are relatively cheaper than `ephSSD`, but offer significantly lower performance. The performance of `persSSD` and `persHDD` scale with volume capacity, whereas `ephSSD` volumes are multiples of 375 GB with a maximum of 4 volumes per VM. Finally, `objStore` is a RESTful object storage service providing the cheapest storage alternative and offering comparable sequential throughput to that of a large `persSSD` volume. Other cloud service providers [13, 10, 21] provide similar storage services with different performance–cost trade-offs.

The heterogeneity in cloud storage services is further complicated by the varying types of jobs within analytics workloads, e.g., iterative applications such as `KMeans` and `Pagerank`, and queries such as `Join` and `Aggregate`. For example, in map-intensive `Grep`, the map phase accounts for the largest part of the execution time (mostly doing I/Os), whereas CPU-intensive `KMeans` spends most of the time performing computation. Furthermore, short-term (within hours) and long-term (daily, weekly or monthly) data reuse across jobs is common in production analytics workloads [41, 32]. As reported in [41], 78% of jobs in Cloudera Hadoop

Table 3.1: Google Cloud storage details. `ephSSD`, `persSSD`, `persHDD` and `objStore` represent VM-local ephemeral SSD, network-attached persistent SSD and HDD, and Google Cloud object storage, respectively.

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

workloads involve data reuse. Another distinguishing feature of analytics workloads is the presence of workflows that represents interdependencies across jobs. For instance, analytics queries are usually converted to a series of batch processing jobs, where the output of one job serves as the input of the next job(s).

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

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

3.1 Background and Related Work

In the following, we categorize and compare previous work with our work on CAST.

Fine-Grained Tiering for Analytics Storage tiering has been studied in the context of data-intensive analytics batch applications. Recent analysis [57] demonstrates that adding a flash tier for serving reads is beneficial for HDFS-based HBase workloads with random

I/Os. As opposed to HBase I/O characteristics, typical MapReduce-like batch jobs issues large, sequential I/Os [91] and run in multiple stages (map, shuffle, reduce). Hence, lessons learned from HBase tiering are not directly applicable to such analytics workloads. hatS [68] and open source Hadoop community [20] have taken the first steps towards integrating heterogeneous storage devices in HDFS for local clusters. However, the absence of task-level tier-aware scheduling mechanisms implies that these HDFS block granularity tiering approaches cannot avoid stragglers within a job, thus achieving limited performance gains if any. PACMan [32] solves this slow-tier straggler problem by using a memory caching policy for small jobs whose footprint can fit in the memory of the cluster. Such caching approaches are complementary to CAST as it provides a coarse-grained, static data placement solution for a complete analytics workload in different cloud storage services.

Cloud Resource Provisioning Considerable prior work has examined ways to automate resource configuration and provisioning process in the cloud. Frugal Cloud File System (FCFS) [86] is a cost-effective cloud-based file storage that spans multiple cloud storage services. In contrast to POSIX file system workloads, modern analytics jobs (focus of our study) running on parallel programming frameworks like Hadoop demonstrate very different access characteristics and data dependencies (described in Section 3.2); requiring a rethink of how storage tiering is done to benefit these workloads. Other works such as Bazaar [60] and Conductor [95], focus on automating cloud resource deployment to meet cloud tenants' requirements while reducing deployment cost. Our work takes a thematically similar view — exploring the trade-offs of cloud services — but with a different scope that targets data analytics workloads and leverages their unique characteristics to provide storage tiering.

Analytics Workflow Optimization A large body of research [100, 75, 51, 79, 72] focuses on Hadoop workflow optimizations by integrating workflow-aware scheduler into Hadoop or interfacing Hadoop with a standalone workflow scheduler. Our workflow enhancement is orthogonal and complements these works as well — CAST++ exploits cloud storage heterogeneity and performance scaling property, and uses opportunities for efficient data placement across different cloud storage services to improve workflow execution. Workflow-aware job schedulers can leverage the data placement strategy of CAST++ to further improve analytics workload performance.

3.2 A Case for Cloud Storage Tiering

3.2.1 Characterization of Data Analytics Workloads

We characterize the analytics workloads along two dimensions. First, we study the behavior of individual applications within a large workload when executed on parallel programming paradigms such as MapReduce — demonstrating the benefits of different storage services for various applications. Second, we consider the role of cross-job relationships (an analyt-

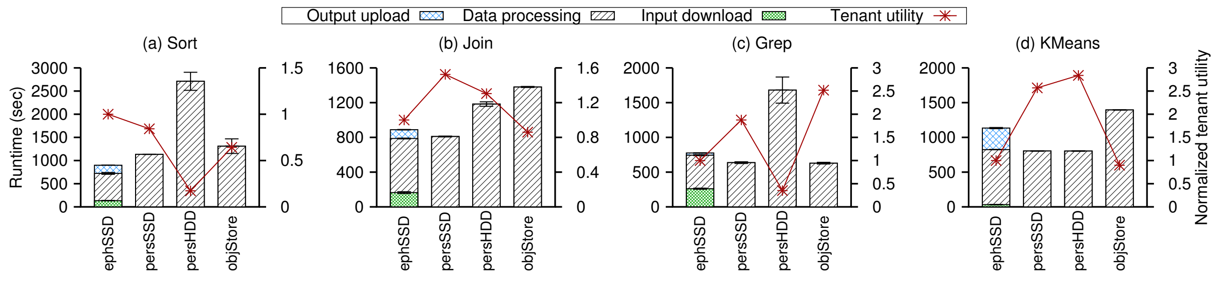
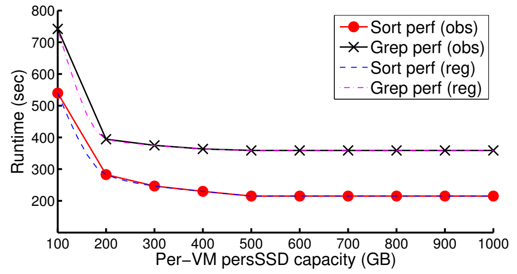


Figure 3.1: Application performance and achieved tenant utility on different cloud storage tiers.

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

(a) Characteristics of studied applications.



(b) Impact of scaling `persSSD` volume capacity for `Sort` and `Grep`. The regression model is used in CAST's tiering approach and is described in detail in 3.3.

Figure 3.2: Characteristics of studied applications on the left. Impact of scaling `persSSD` volume capacity for `Sort` and `Grep` on the right.

ics workload comprises multiple jobs each executing an application) and show how these interactions affect the choice of efficient data placement decisions for the same applications.

Experimental Study Setup

We select four representative analytics applications that are typical components of real-world analytics workloads [41, 101] and exhibit diversified I/O and computation characteristics, as listed in Table 3.2(a). `Sort`, `Join` and `Grep` are I/O-intensive applications. The execution time of `Sort` is dominated by the shuffle phase I/O. `Grep` spends most of its runtime in the map phase I/O, reading the input and finding records that match given patterns. `Join` represents an analytics query that combines rows from multiple tables and performs the join operation during the reduce phase, and thus is reduce intensive. `KMeans` is a CPU-intensive iterative machine learning clustering application. that spends most of its time in the compute phases of map and reduce iterations.

The experiments are performed in Google Cloud using a `n1-standard-16` VM (16 vCPUs, 60 GB memory) with the master node on a `n1-standard-4` VM (4 vCPUs, 15 GB memory). Intermediate data is stored on the same storage service as the original data, except for `objStore`, where we use `persSSD` for intermediate storage.

Analysis: Application Granularity

Figure 3.1 depicts both the execution time of the studied applications and tenant utility for different choices of storage services. We define *tenant utility* (or simply "utility," used interchangeably) to be $\frac{1/_{execution\ time}}{cost\ in\ dollars}$. This utility metric is based on the tenants' economic constraints when deploying general workloads in the cloud. Figure 3.1 (a) shows that `ephSSD` serves as the best tier for both execution time and utility for `Sort` even after accounting for the data transfer cost for both upload and download from `objStore`. This is because there is no data reduction in the map phase and the entire input size is written to intermediate files residing on `ephSSD` that has about 2× higher sequential bandwidth than `persSSD`. On the other hand, Figure 3.1 (b) shows that, `Join` works best with `persSSD`, while it achieves the worst utility on `objStore`. This is due to high overheads of setting up connections to request data transfers using the Google Cloud Storage Connector (GCS connector) for Hadoop APIs [17] for the many small files generated by the involved reduce tasks. Figures 3.1 (c–d) further illustrate the differences of tenant utility and performance between workloads.

Performance Scaling In Google Cloud, performance of network-attached block storage depends on the size of the volume, as shown in Table 3.1. Other clouds be scaled by creating logical volumes by striping (RAID-0) across multiple network-attached block volumes. In Figure 3.2(b), we observe that as the volume capacity increases from 100 GB to 200 GB, the run time of both `Sort` and `Grep` is reduced by 51.6% and 60.2%, respectively. Any further increase in capacity offers marginal benefits. This happens because in both these applications the I/O bandwidth bottleneck is alleviated when the capacity is increased to 200 GB. These observations imply that it is possible to achieve desired application performance in the cloud without resorting to unnecessarily over-provisioning of the storage and thus within acceptable cost.

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

Analysis: Workload Granularity

We next study the impact of cross-job interactions within an analytics workload. To this end, we analyze two typical workload characteristics that have been reported in production

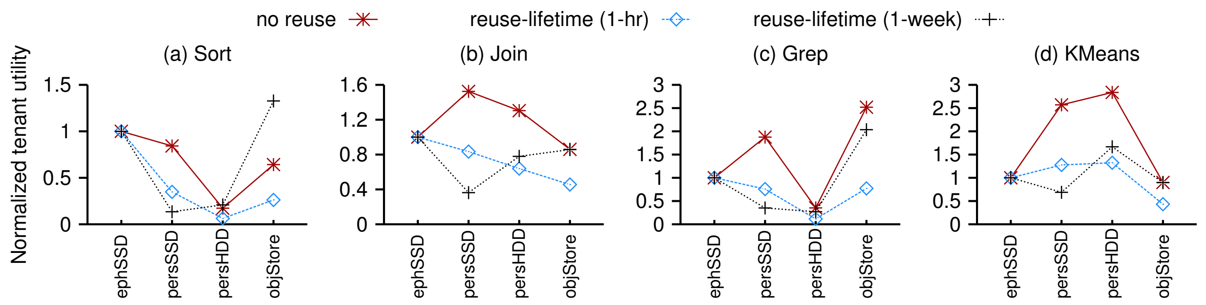


Figure 3.3: Tenant utility under different data reuse patterns. Tenant utility is normalized to that of `ephSSD`.

workloads [41, 32, 82, 57, 49], namely data reuse across jobs, and dependency between jobs, i.e., workflows, within a workload.

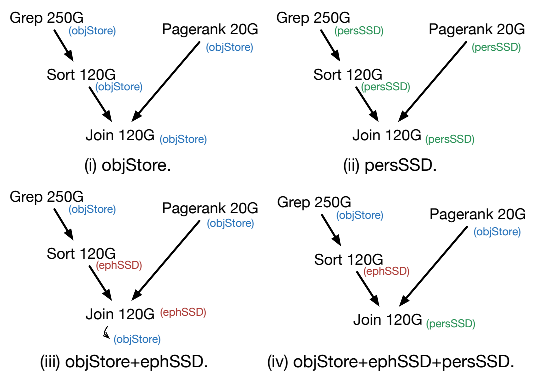
Data Reuse across Jobs As reported in the analysis of production workloads from Facebook and Microsoft Bing [41, 32], both small and large jobs exhibit data re-access patterns both in the short term, i.e., input data shared by multiple jobs and reused for a few hours before becoming cold, as well as in the long term, i.e., input data reused for longer periods such as days or weeks before turning cold. Henceforth, we refer to this characteristic as reuse-lifetime.

Figure 3.3 shows that the choice of storage service changes based on data reuse patterns for different applications. Note that in `reuse-lifetime (1 week)`, data is accessed once per day, i.e., 7 accesses in a week. Similarly, data is accessed once every 8 minutes in `reuse-lifetime (1 hr)`.

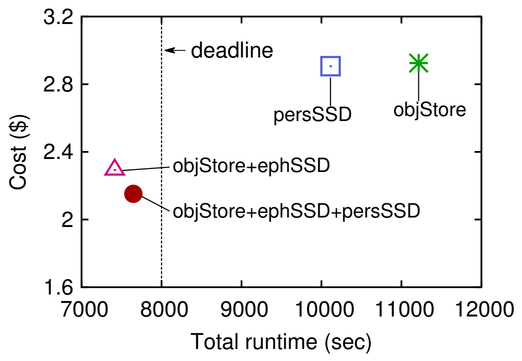
For `ephSSD`, the input download overheads can be amortized by keeping the data in the ephemeral SSD, since the same data will be re-accessed in a very short period of time. This allows `ephSSD` to provide the highest utility for `Join` and `Grep` (Figure 3.3 (b–c)) for `reuse-lifetime (1 hr)`. However, if the data is re-accessed only once per day (`reuse-lifetime (1 week)`), the cost of `ephSSD` far outweighs the benefits of avoiding input downloads. For similar cost reasons, `persSSD`, which demonstrates the highest utility for individual applications (Figure 3.1 (b)), becomes the worst choice when long term re-accesses are considered. Furthermore, as expected, the behavior of CPU-intensive `KMeans` (Figure 3.3 (d)) remains the same across reuse patterns.

Workflows in an Analytics Workload An analytics workflow consists of a series of jobs with inter-dependencies, as a Directed Acyclic Graph (DAGs) of actions [7]. For our analysis, we consider the workflows where the output of one job acts as a part of an input of another job.

Consider the following example. Figure 3.4(a) lists four possible tiering plans for a four-job workflow. Figure 3.4(a) (i) and Figure 3.4(a) (ii) depict cases where a single storage service,



(a) Four possible workflow tiering plans.



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

Figure 3.4: Possible tiering plans for a simple 4-job workflow. Storage medium in parentheses indicates where the input of the job is stored.

`objStore` and `persSSD`, respectively, is used for the entire workflow. As shown in Figure 3.4(b), these two data placement strategies not only perform poorly (missing a hypothetical deadline of 8,000 seconds) but also results in high costs compared to the other two hybrid storage plans. On the other hand, both the hybrid storage services meet the deadline and perform better.

Key Insights (i) Not only do data analytics workloads require use of different storage services for different applications, the data placement choices also change when data reuse effects are considered. (ii) Use of multiple criteria by the tenant, such as workflow deadlines and monetary costs, adds more dimensions to a data placement planner and requires careful thinking about tiering strategy.

3.3 CAST Framework

We build CAST, an analytics storage tiering framework that exploits heterogeneity of both the cloud storage and analytics workloads to satisfy the various needs of cloud tenants. Furthermore, CAST++, an enhancement to CAST, provides data pattern reuse and workflow awareness based on the underlying analytics framework. Figure 3.5 shows the high-level overview of CAST operations and involves the following components. (1) The analytics *job performance estimator* module evaluates jobs execution time on different storage services using workload specifications provided by tenants. These specifications include a list of jobs, the application profiles, and the input data sizes for the jobs. The estimator combines this with compute platform information to estimate application run times on different storage services. (2) The *tiering solver* module uses the job execution estimates from the *job performance estimator* to generate a tiering plan that spans all storage tiers on the specific

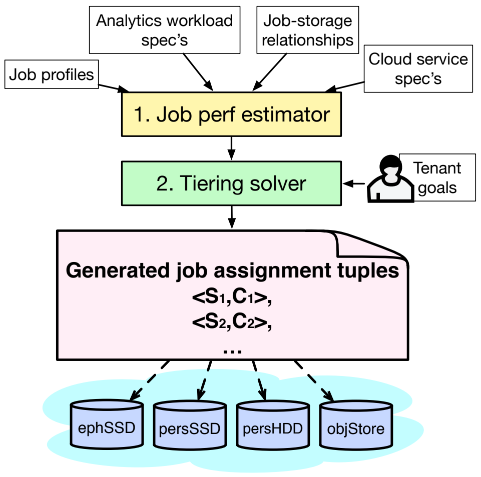


Figure 3.5: Overview of CAST tiering framework.

cloud provider available to the tenant. The objective of the solver is to satisfy the high-level tenants' goals such as achieving high utility or reducing deadline miss rates.

3.3.1 Estimating Analytics Job Performance

$$EST(\hat{\mathcal{R}}, \hat{\mathcal{M}}(s_i, \hat{\mathcal{L}}_i)) = \overbrace{\underbrace{\left\lceil \frac{m}{n_{vm} \cdot m_c} \right\rceil}_{\#\ \text{waves}} \cdot \underbrace{\left(\frac{input_i/m}{bw_{map}^{s_i}} \right)}_{\text{Runtime per wave}}}^{\text{map phase}} + \underbrace{\left\lceil \frac{r}{n_{vm} \cdot r_c} \right\rceil \cdot \left(\frac{inter_i/r}{bw_{shuffle}^{s_i}} \right)}_{\text{shuffle phase}} + \underbrace{\left\lceil \frac{r}{n_{vm} \cdot r_c} \right\rceil \cdot \left(\frac{output_i/r}{bw_{reduce}^{s_i}} \right)}_{\text{reduce phase}} \tag{3.1}$$

We leverage and adapt MRCute [60] model in CAST to predict job execution time, due to its ease-of-use, availability, and applicability to our problem domain. Equation 3.1 defines our performance prediction model. It consists of three sub-models — one each for the map, shuffle, and reduce phases — where each phase execution time is modeled as *#waves* × *runtime per wave*. A *wave* represents the number of tasks that can be scheduled in parallel based on the number of available slots. CAST places all the data of a job on a single storage service with predictable performance, and tasks (within a job) work on equi-sized data chunks. The estimator also models wave pipelining effects — in a typical MapReduce execution flow, the three phases in the same wave are essentially serialized, but different waves can be overlapped. Thus, the prediction model does not sacrifice estimation accuracy. Table 3.2 lists the notations used in the model.

3.3.2 Basic Tiering Solver

The basic tiering solver uses a simulated annealing algorithm [66] to systematically search through the solution space and find a desirable tiering plan, given the workload specification, analytics models, and tenants' goals.

Table 3.2: Notations used in the analytics jobs performance prediction model and CAST tiering solver.

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

CAST Solver: Model

The data placement and storage provisioning problem is modeled as a non-linear optimization problem that maximizes the tenant utility (U) defined in Equation 3.2:

$$max \;\; U = \frac{{}^{1}/_{T}}{(\$_{vm} + \$_{store})} \;\;, \tag{3.2}$$

$$s.t. \;\; c_i \geq (input_i + inter_i + output_i) \; (\forall i \in J) \;\;, \tag{3.3}$$

$$T = \sum_{i=1}^{J} REG\big(s_i, capacity[s_i], \hat{\mathcal{R}}, \hat{\mathcal{L}_i}\big) \;, where \;\; s_i \in F \;, \tag{3.4}$$

$$\$_{vm} = n_{vm} \cdot (price_{vm} \cdot T) \;, \tag{3.5}$$

$$\$_{store} = \sum_{f=1}^{F} \Big(capacity[f] \cdot (price_{store}[f] \cdot \big\lceil {}^{T}/_{60} \big\rceil)\Big) \tag{3.6}$$

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

The performance is modeled as the *reciprocal* of the estimated completion time in minutes $({}^{1}/_{T})$ and the costs include both the VM and storage costs. Equation 3.3 defines the *capacity constraint*, which ensures that the storage capacity (c_i) provisioned for a job is sufficient to meet its requirements for all the phases (map, shuffle, reduce). We also consider intermediate data when determining aggregated capacity. Given a specific tiering solution, the estimated total completion time of the workload is defined by Equation 3.4. Since job performance

Algorithm 1: Simulated Annealing Algorithm.

```
Input:  Job information matrix: $\hat{\mathcal{L}}$ ,
          Analytics job model matrix: $\hat{\mathcal{M}}$ ,
          Runtime configuration: $\hat{\mathcal{R}}$ ,
          Initial solution: $\hat{P}_{init}$ .
Output: Tiering plan $\hat{P}_{best}$
begin
    $\hat{P}_{best} \leftarrow \{\}$
    $\hat{P}_{curr} \leftarrow \hat{P}_{init}$
    $exit \leftarrow False$
    $iter \leftarrow 1$
    $temp_{curr} \leftarrow temp_{init}$
    $U_{curr} \leftarrow Utility(\hat{\mathcal{M}}, \hat{\mathcal{L}}, \hat{P}_{init})$
    while not exit do
        $temp_{curr} \leftarrow Cooling(temp_{curr})$
        for next $\hat{P}_{neighbor}$ in $AllNeighbors(\hat{\mathcal{L}}, \hat{P}_{curr})$ do
            if $iter > iter_{max}$ then
                $exit \leftarrow True$
                break
            $U_{neighbor} \leftarrow Utility(\hat{\mathcal{M}}, \hat{\mathcal{L}}, \hat{P}_{neighbor})$
            $\hat{P}_{best} \leftarrow UpdateBest(\hat{P}_{neighbor}, \hat{P}_{best})$
            $iter{+}{+}$
            if $Accept(temp_{curr},\ U_{curr},\ U_{neighbor})$ then
                $\hat{P}_{curr} \leftarrow \hat{P}_{neighbor}$
                $U_{curr} \leftarrow U_{neighbor}$
                break
    return $\hat{P}_{best}$
```

in the cloud scales with capacity of some services, we use a third degree polynomial-based cubic Hermite spline [67] regression model, $REG(s_i,\ .)$, to estimate the execution time. In every iteration of the solver, the regression function uses the storage service (s_i) assigned to a job in that iteration, the total provisioned capacity of that service for the entire workload, cluster information such as number of VMs and the estimated runtime based on Equation 3.1 as parameters. We show the accuracy of the splines in Figure 3.2(b).

CAST Solver: Simulated Annealing-based Algorithm

The algorithm (Algorithm 1) takes as input workload information $(\hat{\mathcal{L}})$, compute cluster configuration $(\hat{\mathcal{R}})$, and information about performance of analytics applications on different storage services $(\hat{\mathcal{M}})$ as defined in Table 3.2. Furthermore, the algorithm uses $\hat{P}_{init}$ as the initial tiering solution that is used to specify preferred regions in the search space. For example, the results from the characteristics of analytics applications described in Table 3.2(a) can be used to devise an initial placement.

3.3.3 Enhancements: CAST++

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

Enhancement 1: Data Reuse Pattern Awareness To incorporate data reuse patterns across jobs, CAST++ ensures that all jobs that share the same input dataset have the same storage service allocated to them, as captured by Constraint 3.7.

$$s_i \equiv s_l \; (\forall i \in D, ß \neq l, \in D) \tag{3.7}$$
$$where \;\; D = \{all \;\; jobs \;\; which \;\; share \;\; input\}$$

Enhancement 2: Workflow Awareness Prior research has shown that analytics workflows are usually associated with a tenant-defined deadline [72, 51]. For workloads with a mix of independent and inter-dependent jobs, the basic dependency-oblivious CAST may either increase the deadline miss rate or unnecessarily increase the costs, as we show in 3.2.1. To this end, we enhance the basic solver to consider the objective of minimizing the total monetary cost (Equation 3.8) and introduce a constraint to enforce that the total estimated execution time meets the predefined deadline (Equation 3.9).

$$min \;\; \$_{total} = \$_{vm} + \$_{store} \; , \tag{3.8}$$
$$s.t. \;\; \sum_{i=1}^{J_w} REG\Big(s_i, s_{i+1}, capacity[s_i], \hat{\mathcal{R}}, \hat{\mathcal{L}}\Big) \leq deadline \; , \tag{3.9}$$
$$c_i \geq \sum_{i=1}^{J_w} \big((s_{i-1} \neq s_i) \cdot input_i + inter_i \tag{3.10}$$
$$+ (s_{i+1} \equiv s_i) \cdot output_i\big) \; , where \; s_0 = \phi \; .$$

Furthermore, Equation 3.10 restricts the capacity constraint in Equation 3.3 by incorporating inter-job dependencies. The updated approach only allocates capacity if the storage tier for the output of a job at the previous level is not the same as input storage tier of a job at the next level in the DAG. To realize this approach, we enhance Algorithm 1 by replacing the next neighbor search ($AllNeighbors(.)$) with a depth-first traversal in the workflow DAG. This allows us to reduce the deadline miss rate.

3.4 Evaluation

In this section, we present the evaluation of CAST and CAST++ using a 400-core Hadoop cluster on Google Cloud. Each slave node in our testbed runs on a 16 vCPU `n1-standard-16`

Table 3.3: Distribution of job sizes in Facebook traces and our synthesized workload.

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

VM as specified in 3.2 with a 100-job analytics workload.

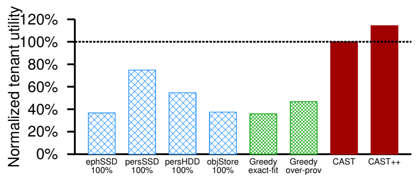
3.4.1 Tenant Utility Improvement

Methodology

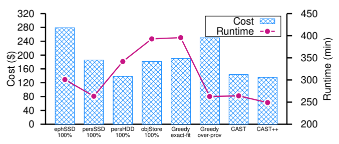
We compare CAST against six storage configurations: four without tiering and two that employ a greedy algorithm for static tiering. For completeness, we compare our approach with two versions of the greedy algorithm: `Greedy exact-fit` attempts to limit the cost by not over-provisioning extra storage space for workloads, while `Greedy over-provisioned` will assign extra storage space as needed to reduce the completion time and improve performance. We generate a representative 100-job workload by sampling the input sizes from the distribution observed in production traces from a $3,000$-machine Hadoop deployment at Facebook [41]. We quantize the job sizes into 7 bins as listed in Table 3.3, to enable us to compare the dataset size distribution across different bins. The largest job in the Facebook traces has $158,499$ map tasks. Thus, we choose $3,000$ for the highest bin in our workload to ensure that our workload demands a reasonable load but is also manageable for our 400-core cluster. Since there is a moderate amount of data reuse throughout the Facebook traces, we also incorporate this into our workload by having 15% of the jobs share the same input data. We assign the four job types listed in Table 3.2(a) to this workload in a round-robin fashion to incorporate the different computation and I/O characteristics.

Effectiveness for General Workload

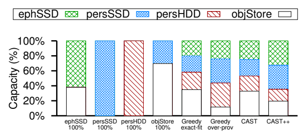
Figure 3.6 shows the results for tenant utility, performance, cost and storage capacity distribution across four different storage services. We observe in Figure 3.6(a) that CAST improves the tenant utility by 33.7% – 178% compared to the configurations with no explicit tiering, i.e., `ephSSD 100%`, `persSSD 100%`, `persHDD 100%` and `objStore 100%`. The best combination under CAST consists of 33% `ephSSD`, 31% `persSSD`, 16% `persHDD` and 20% `objStore`,



(a) Normalized tenant utility.



(b) Total cost and runtime.



(c) Capacity breakdown.

Figure 3.6: Effectiveness of CAST and CAST++ on workloads with reuse, observed for key storage configurations. Tenant utility is normalized to that of the configuration from basic CAST.

as shown in Figure 3.6(c). `persSSD` achieves the highest tenant utility among the four non-tiered configurations, because `persSSD` is relatively fast and persistent. The tenant utility of `Greedy exact-fit` is as poor as `objStore 100%`. This is because `Greedy exact-fit` only allocates *just enough* storage space without considering performance scaling. On the other hand, `Greedy over-provisioned` significantly over-provisions `persSSD` and `persHDD` space to improve the runtime of the jobs. The tenant utility improvement under basic CAST is 178% and 113.4%, compared to `Greedy exact-fit` and `Greedy over-provisioned`, respectively.

Effectiveness for Data Reuse

CAST++ outperforms all other configurations and further enhances the tenant utility of basic CAST by 14.4% (Figure 3.6(a)). This is due to the following reasons. (1) CAST++ successfully improves the tenant utility by exploiting the characteristics of jobs and underlying tiers and tuning the capacity distribution. (2) CAST++ effectively detects data reuse across jobs to further improve the tenant utility by placing shared data in the fastest `ephSSD`, since we observe that in Figure 3.6(c) the capacity proportion under CAST++ of `objStore` reduces by 42% and that of `ephSSD` increases by 29%, compared to CAST. This is because CAST++ places jobs that share the data on `ephSSD` to amortize the data transfer cost from `objStore`.

3.4.2 Meeting Workflow Deadlines

In our next set of experiments, we evaluate the ability of CAST++ to meet workflow deadlines while minimizing cost. We compare CAST++ against four storage configurations without tiering and a fifth configuration from the basic, workflow-oblivious CAST. This experiment employs five workflows with a total of 31 analytics jobs, with the longest workflow consisting of 9 jobs.

We consider the completion time between the start of its first job and the completion of its last job. The *deadline* of a workflow is the upper limit on this completion time. We set the deadline of the workflows between 15 – 40 minutes based on the job input sizes and the job

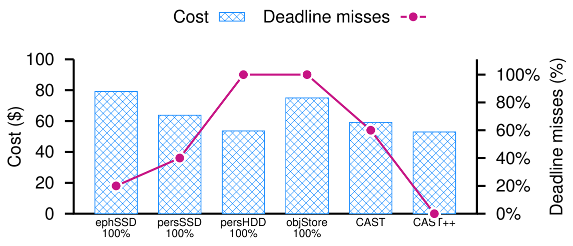


Figure 3.7: Deadline miss rate and cost of CAST++ compared to CAST and four non-tiered configurations.

types comprising each workflow. Note that the time taken for this cross-tier between jobs transfer is accounted as part of the workflow runtime by CAST++. Figure 3.7 shows the *miss rate* of workflow deadlines for the studied configurations. CAST++ meets all the deadlines and incurs the lowest cost, comparable to that of `persHDD` that is the lowest-priced but the slowest tier and has a miss rate of 100%. CAST misses 60% of the deadlines because it optimizes tenant utility for each job individually and does not account for cross-tier transfer time. As shown in Figure 3.7, CAST++ also non-tiered storage configurations in meeting workflow deadlines.

Chapter 4

Provider versus Tenant: Pricing Games for Hybrid Object Stores in the Cloud

Cloud providers continue to use low-cost HDDs as the underlying storage medium for their cloud-object storage deployments. This is because the price gap between HDDs and SSDs continues to be significant, especially for datacenter-scale deployments. Object stores have traditionally been used as data dumps for large objects such as backup archives and large-volume pictures or videos; use cases where SSDs would incur a high acquisition as well as maintenance cost [84], e.g., premature device replacement. Nevertheless, recent research has shown that SSDs can deliver significant benefits for many types of Big Data analytics workloads [45, 57]. Even newer, promising technology on this front does not adequately address the cost and performance trade-offs. For example, while the 3-bit MLC NAND technology promises to deliver higher SSD densities and potentially drive down the acquisition cost, it has taken a major toll on SSD device lifetime [55, 78, 97].

As we have shown in Chapter 3, data analytics applications are particularly amenable to tiered storage deployments because of the inherent heterogeneity in workload I/O patterns. Therefore, we argue that traditional HDD-based object stores are inefficient for data analytics. For cloud tenants, an HDD-based object store cannot effectively meet their requirements (e.g., deadlines) due to the relatively slow I/O performance of HDDs. For cloud provider, an HDD-only object store does not provide any pricing leverage, which reduces profitability. A faster tier can provide a higher quality-of-service (QoS), which can be strategically priced to increase profits. Hence, a hybrid HDD–SSD approach is desirable for both cloud providers and tenants.

To this end, we propose an innovative tiered object store that exposes this tiering control to tenants. Thus, the tenants can meet their price–performance objectives by partitioning their workloads to utilize different tiers based on their application characteristics. The cloud provider offers the tiers under dynamic pricing based on tenant's usage of each tier enabling it to offset the increased management costs of faster tiers.

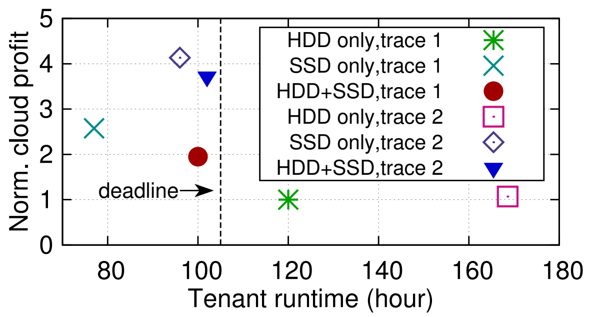


Figure 4.1: Tenant workload runtime for `trace 1` and `trace 2`, and provider's profits under different configurations. Cloud profits normalized to that of `HDD only,trace 1`.

To verify our argument, we conducted a trace-driven simulation study by replaying two 250-job (job types described in 3.4) snippet traces from a larger set of traces from Facebook's production Hadoop cluster [41] similar to the one used in Chapter 3. We set the HDD tier price as and the SSD tier price as \$0.0044/GB/day, i.e., 4× the HDD price. `trace 1` consumes 12 TB data and generates 4.7 TB output, while `trace 2` consumes 18 TB data and generates 8.2 TB output. For the hybrid storage tiering case (`HDD+SSD`), the tenant places jobs in different tiers with a desired workload completion deadline of 105 hours. For this purpose, we use Algorithm 2 that optimizes the tier allocation to meet the deadline while minimizing the cost (4.1.2). Our simulation assumes that tenant's SSD allocation is not limited by the cloud provider's SSD capacity.

Figure 4.1 shows the results, from which we make three observations. (1) Workloads with `HDD-only` config. cannot meet the tenant-defined deadline and the cloud provider earns the lowest profit. (2) With `HDD+SSD` tiering config., both workloads are able to meet the deadline, while the cloud provider sees significantly more profit (`trace 2` has larger input and output datasets and hence yields more profit). This is because the tenant places part of the workloads on the SSD tier, which is more expensive than the HDD tier. (3) `SSD only` config. improves performance, but with marginally higher profit, compared to `HDD+SSD`. This is mainly due to `HDD+SSD`'s tiering optimization. This experiment demonstrates that through object storage tiering both the cloud provider and tenants can effectively achieve their goals.

Cloud providers have a multitude of device options available for deploying object storage infrastructure, including HDDs with different RPMs, SATA and PCIe SSDs and the emerging SCM devices, etc. Each device type offers different cost/GB and, in case of SSDs and SCMs, different endurance. Therefore, estimating price points while keeping maintenance costs under control is a challenge. For example, while the cloud providers might want to encourage tenants to use more SSDs to increase their profits, they run the risk of SSD wear-out earlier than expected resulting in increasing osts and decreasing overall profits. To remedy this issue, we introduce a dynamic pricing model that providers can leverage to mitigate additional costs and increase overall operating profits for providers. The dynamic pricing model has two objectives: (1) to balance the price-increase and SSD wear-out rate by exploiting the trade-off between high revenue versus high operational costs for *high profit*;

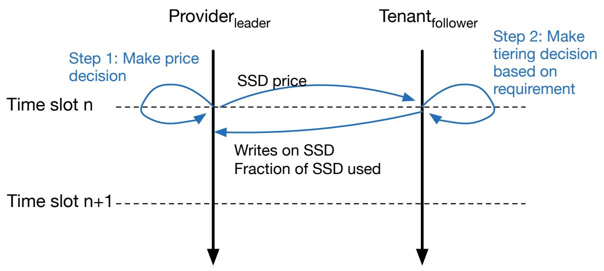


Figure 4.2: Illustration of the leader/follower game. The two entities repeat the same sequence of events on time $n + 1$.

(2) to provide an effective incentivizing mechanism to tenants so that the tenants can meet their goals via object store tiering in a more cost-efficient fashion.

The novelty of this work is that we turn the pricing of object storage in to a game, placing the cloud provider in the role of the *leader*, and it decides prices for the offered storage tiers. The tenant *follows* this decision by deciding upon how much of each tier to use, such that its deadlines can be met in the minimum possible cost. The provider starts the next move by adjusting its prices based on the usage of each tier by the client in the previous move, in a way that maximizes its profit. Hence, *we take the first step towards providing such a cloud-provider-driven game-theoretic pricing model through object storage tiering.*

4.1 Model Design

We design a leader/follower cloud pricing framework with the objective of maximizing a cloud provider's profit. In our model, the game is played in two steps (Figure 4.2). First, the cloud provider (leader) makes the pricing decisions (4.1.1) based on predictions of tenants' demand on storage resources. Second, given prices of different storage resources, a tenant (follower) makes tiering decisions[1] based on her own requirements (4.1.2), and the strategy is represented by the tenant's storage tiering specification, i.e., which jobs use what tier.

While the tenants can see the price changes by the provider, they are unaware of the actual reasons for the changes. Hence, in our formulation tenants can only predict the provider's price movements based on historical data. Similarly, the provider is not aware of explicit tenant requirements, and only garners information from the requested storage capacity and the writes operations. Thus, the provider also only uses historical information about these aspects to predict tenant demand. Consequently, both the tenants and the provider models adopted in our game are purposefully "myopic" controls for predicting only the next time slot, and not beyond that in the future.

[1] In our current tenant model, we assume: (1) tenants only make tiering decisions for the next time slot; and (2) adjustment for previously launched workloads due to price changes is not necessarily needed as workload per slot can be finished within that particular slot.

4.1.1 Provider Model

We model the provider cost as follows. Assuming that the fraction of the cost that comes from SSDs wear-out is $t < 1$,[2] the cost can be modeled as: $cost = \frac{1}{t} \cdot \frac{p_{ssd}}{endurance} \cdot w$, where p_{ssd} is the market price of one SSD, *endurance* is the endurance lifespan of the particular SSD, and w is the amount of data written to the SSD in GB. The pricing decision making process can be modeled as a non-linear optimization problem that maximizes the profit defined as $profit = \sum_i \Big(\sum_f (capacity_f \cdot f_{(i,f)} \cdot p_{(i,f)}) - cost_i \Big)$, where i is the time unit index and f is the storage service.

In a time slot n, we predict the SSD demand proportion for the next time slot $n+1$ ($f_{(n+1,ssd)}$), which depends on the difference between the predicted SSD price for $n+1$ and the calculated SSD price for n. The predicted HDD demand proportion is in turn determined by the total storage demand subtracting the predicted SSD demand proportion.

The amount of data that will be written to SSDs is determined by the difference of predicted SSD demand proportion in time slot $n + 1$ to that in time slot n. If the SSD demand is predicted to increase, it implies that the amount of data that will be absorbed by the SSD tier will also increase. Our provider model also enforces a SSD tier data writing constraint, which ensures that the expected amount of data written to the SSD tier will not exceed the threshold that is calculated based on accumulated historical statistics. The factor indirectly controls the value adaptation of decision variables $p_{(n,b)}$ and $p'_{(n+1,b)}$. We assume HDD prices $p_{(n,a)}$ and $p_{(n+1,a)}$ are fixed, and SSD prices are constrained in a pre-defined range.

4.1.2 Tenant Model

The data placement and storage provisioning at the tenant side is modeled as a non-linear optimization problem as well. The goal is to maximize *tenant utility*, which is defined as $utility = \frac{1}{(T \cdot \$)}$ (where T is the total workload runtime and \$ is the total monetary cost, refer to Chapter 3 for details). A *capacity constraint* is required to ensure that the storage capacity provisioned for a job is sufficient to meet its requirements for all the workload phases (map, shuffle, reduce). Given a specific tiering solution, the estimated total completion time of the workload is constrained by a tenant-defined *deadline*. A price predictor at the tenant side is needed, the value of which can also be supplied as a hint by the cloud provider.

We devise a simulated annealing based algorithm (Algorithm 2) for computing tenants' data partitioning and job placement plans. The main goal of our algorithm is to find a near-optimal tiering plan for a given workload. The algorithm takes as input workload information ($\hat{\mathcal{L}}$), compute cluster configuration ($\hat{\mathcal{R}}$), and information about performance of analytics applications on different storage services ($\hat{\mathcal{M}}$). $\hat{P}_{init}$ serves as the initial tiering

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

Algorithm 2: Tiering solver.

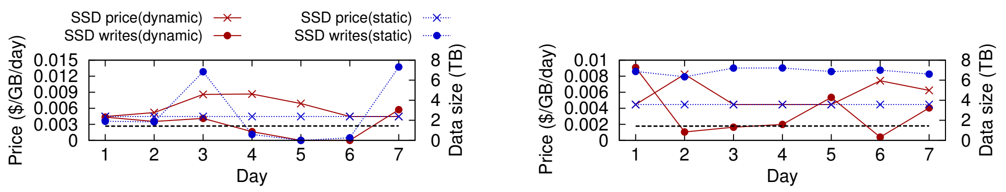
```
Input: Job information matrix: $\hat{\mathcal{L}}$,  Analytics job model matrix: $\hat{\mathcal{M}}$,  Runtime configuration: $\hat{\mathcal{R}}$,
       Initial solution: $\hat{P}_{init}$.
Output: Tiering plan $\hat{P}_{best}$
begin
    $\hat{P}_{best} \leftarrow \{\}$
    $\hat{P}_{curr} \leftarrow \hat{P}_{init}$
    $exit \leftarrow False$
    $iter \leftarrow 1$
    $temp_{curr} \leftarrow temp_{init}$
    $U_{curr} \leftarrow Utility(\hat{\mathcal{M}}, \hat{\mathcal{L}}, \hat{P}_{init})$
    while $not\ exit$ do
        $temp_{curr} \leftarrow Cooling(temp_{curr})$
        for $next\ \hat{P}_{neighbor}\ in\ AllNeighbors(\hat{\mathcal{L}}, \hat{P}_{curr})$ do
            if $iter > iter_{max}$ then
                $exit \leftarrow True$
                break
            $U_{neighbor} \leftarrow Utility(\hat{\mathcal{M}}, \hat{\mathcal{L}}, \hat{P}_{neighbor})$
            $\hat{P}_{best} \leftarrow UpdateBest(\hat{P}_{neighbor}, \hat{P}_{best})$
            $iter{+}{+}$
            if $Accept(temp_{curr},\ U_{curr},\ U_{neighbor})$ then
                $\hat{P}_{curr} \leftarrow \hat{P}_{neighbor}$
                $U_{curr} \leftarrow U_{neighbor}$
                break
    return $\hat{P}_{best}$
```

solution that is used to specify preferred regions in the search space. The results from a simple greedy algorithm based on the characteristics of analytics applications (e.g., the four described in 3.4) can be used to devise an initial placement. The full details of the tenant model can be found in Chapter 3 and [60].

4.2 Evaluation

We have used trace-driven simulations to demonstrate how our cloud–tenant interaction models perform in practice. It simulates an object store with built-in tiering mechanism that tenants can control. We use the production traces collected from a $3,000$-machine Hadoop deployment at Facebook [41]. The original traces consist of $25,428$ Hadoop jobs; we chose to use a snippet of $1,750$ jobs that span a 7-day period. We set the time slot for our models to one day. We assign to our workload, in a round-robin fashion, four representative analytics applications that are typical components of real-world analytics [41, 101] and exhibit diversified I/O and computation characteristics [45]: `Sort`, `Join`, `Grep` and `KMeans` (for more details, see Section 3.2.1.

Figure 4.3 shows price variation by the cloud provider's model based on the amount of data written by the tenant to the SSD tier on a per-day basis. The HDD price is fixed, while the

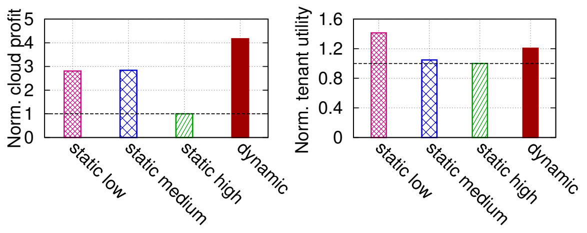


(a) A 7-day trace. (b) A single-day trace repeated 7 times.

Figure 4.3: Dynamic pricing by the provider and the tenant's response for two 7-day workloads. The dotted horizontal line represents the daily write limit L_i. The variation in L_i is too small to be discernible at this scale.

SSD price is dynamically adjusted on a daily basis for `dynamic` pricing (the pricing is the same as for Figure 4.1). Under `static` pricing, the provider sets a static price and tenants periodically run Algorithm 2, whereas under `dynamic` pricing, the provider and tenants interact. The per-day write limit $\mathcal{L}_n$ is dynamically adjusted based on the amount of writes from the tenant side (though not discernible in the figure). Figure 4.3(a) shows the price changes for a 7-day trace, with a different workload running on each day. We observe that as the amount of writes by the tenant on SSD tier increases above the write limit, the cloud provider begins to adjust the SSD price. The tenant's model will adjust the tiering strategy to either put more data in the HDD tier or pay more for the SSD tier in the case of a strict deadline requirement. Since, each day has a different workload and hence a different deadline.

Figure 4.3(b) shows the price changes for a 7-day period with the same single-day trace replayed every day. This trace shows stronger correlation between per-day SSD writes from the tenant and the SSD price from the provider. This workload exhibits the same specifications every day (e.g., dataset size, a relaxed deadline, etc.), thus the daily writes on the SSD tier remain stable under static pricing. However, the dynamic pricing model can effectively respond to the spike in the amount of writes on the first day and adjust the SSD price accordingly. Given the increased SSD price, the tenant tries to reduce their monetary cost by migrating more jobs to the cheaper HDD tier, while still meeting the deadline. This is, in turn, implicitly controlled by cloud provider's prediction model. The provider is aware of the total storage demand (behavior patterns) from the tenant (this is why we chose to repeat a single day trace) and hence, by adjusting SSD pricing, the overall writes to the SSD tier is maintained within a reasonably smaller range (which is as desired by provider but will not break the relatively relaxed deadline constraint imposed by tenant). When the provider lowers the SSD price in response, the tenant increases their use of SSD, prompting the provider to increase the SSD price again. This interaction of the tenant and the provider results in an average of 2.7 TB/day SSD writes compared to an average of 7 TB/day under static pricing (with 0% deadline miss rate for both cases). The test demonstrates that our dynamic pricing model can adaptively adjust the SSD pricing based on the amount of data written to the SSD tier to maintain high profits while keeping the SSD wear-out under control by keeping write loads to SSD in check.



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

Figure 4.4: Cloud profit and tenant utility averaged over 7 days. `static low`, `static medium` and `static high` mean a low, medium, and high static SSD price, respectively. Cloud profit and tenant utility are normalized wrt. `static high`. Results are normalized to that of `static high`.

In our next test, we examine the impact of different SSD pricing models on provider profit and tenant utility, i.e., the cloud–tenant interaction. Figure 4.4 shows the results. We choose three static prices for SSDs: low (the minimum SSD price that we use: \$0.0035/GB/day), medium (\$0.0082/GB/day), and high (the maximum SSD price, \$0.0121/GB/day). We also compare static prices with our dynamic pricing model. As observed in Figure 4.4(a), dynamic pricing yields the highest provider profit as it increases the price based on SSD writes from the tenant. Both `static low` and `medium high` yield similar profits that are 32.6% lower than that gained under dynamic pricing. This is because `static medium` results in more jobs placed on the HDD tier, which lowers the tenant cost while causing longer workload completion time. `static low` is not able to generate enough profit due to the very low price, while under `static high` the tenant solely migrates all the jobs to the HDD tier, thus resulting in low profit.

Next, we examine the tenant utility in Figure 4.4(b). With `static low` SSD price, the tenant utility is 17.1% higher than that achieved under dynamic pricing. However, this would result in significantly shortened SSD lifetime (by as much as 76.8%), hence hurting the cloud profit in the long term. With `static high` SSD price, the tenant utility is reduced by 17.1% compared to that of dynamic pricing, as the tenant shifts most jobs to the HDD tier. `static medium` SSD price yields slightly higher tenant utility as compared to `static high` but still 13.1% lower than that seen under dynamic pricing. This is because the tenant has to assign some jobs to the faster SSD tier in order to guarantee that the workload does not run for too long. Dynamic pricing, on the other hand, maintains the tenant utility at a reasonably high level (higher than both `static medium` and `static high` but slightly lower than `static low`), while guaranteeing that the SSD lifetime constraints are met. This demonstrates that our dynamic pricing model can effectively achieve a win–win for both the cloud provider and tenants.

4.3 Related Work

Storage Tiering Researchers demonstrates that adding a SSD tier for serving reads is beneficial for HDFS-based HBase and Hadoop [57]. Existing implementations of cloud object stores provide mechanisms for tiered storage. Ceph, which exposes an object store API, has also added tiering support [11]. The cost model and tiering mechanism used in traditional storage tiering approaches [56, 57] cannot be directly applied to analytics batch processing applications running in a public cloud environment, mainly due to cloud storage and analytics workload heterogeneity. Our work focuses on providing insights into the advantages of dynamically priced tiered object storage management involving both cloud providers and tenants.

Dynamic Pricing Researchers have also looked at cloud dynamic pricing [76, 90]. CRAG [12] focuses on solving the cloud resource allocation problems using game theoretical schemes, while Londono et al. [77] propose a cloud resource allocation framework using colocation game strategy with static pricing. Ben-Yehuda et al. [29] propose a game-theoretic market-driven bidding scheme for memory allocation in the cloud. In the context of network pricing, Sen et al. [89] study a wide variety of pricing and incentivizing mechanisms for solving the network congestion problems. We adopt a simplified game theoretic model where the cloud providers give incentives in the form of dynamic pricing and tenants adopt tiering in object stores for achieving their goals.

4.4 Summary

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

Chapter 5

Bibliography

[1] AccuSim: Accurate Simulation of Cache Replacement Algorithms. https://engineering.purdue.edu/~ychu/accusim/.

[2] Amazon EMR. http://aws.amazon.com/elasticmapreduce.

[3] Amazon Glacier. https://aws.amazon.com/glacier/.

[4] AmazonS3 Connector for Hadoop. https://wiki.apache.org/hadoop/AmazonS3.

[5] Android phones. http://www.android.com/phones/.

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

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

[8] Apple iPhone. https://apple.com/iphone/.

[9] AWS Case Studies. https://aws.amazon.com/solutions/case-studies/.

[10] Azure Storage. http://azure.microsoft.com/en-us/services/storage.

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

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

[13] EC2 Storage. http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/Storage.html.

[14] Filebench. http://filebench.sourceforge.net/.

[15] Fitbit. http://www.fitbit.com/.

[16] Flashcache. `https://github.com/facebookarchive/flashcache`.

[17] Google Cloud Storage Connector for Hadoop. `https://cloud.google.com/hadoop/google-cloud-storage-connector`.

[18] Google Glass. `https://www.google.com/glass/start/`.

[19] Hadoop on Google Compute Engine. `https://cloud.google.com/solutions/hadoop`.

[20] HDFS-2832. `https://issues.apache.org/jira/browse/HDFS-2832`.

[21] HP Cloud Storage. `http://www.hpcloud.com/products-services/storage-cdn`.

[22] Intel 3D Xpoint. `https://www.intel.com/content/www/us/en/architecture-and-technology/intel-micron-3d-xpoint-webcast.html`.

[23] Microsoft Azure HDInsight. `http://azure.microsoft.com/en-us/services/hdinsight`.

[24] MRAM Companies. `https://www.mram-info.com/companies`.

[25] SNIA: IOTTA Repository Home. `http://iotta.snia.org/traces/388`.

[26] Specs for first Intel 3D XPoint SSD. `https://arstechnica.com/information-technology/2017/02/specs-for-first-intel-3d-xpoint-ssd-so-so-transfer-speed-awesome-random-io/`.

[27] Strace. `http://linux.die.net/man/1/strace`.

[28] Two headed monster. `https://en.wikipedia.org/wiki/Two-Headed_Monster`.

[29] Agmon Ben-Yehuda, O., Posener, E., Ben-Yehuda, M., Schuster, A., and Mu'alem, A. Ginseng: Market-driven memory allocation. In *ACM VEE* (2014).

[30] Akyürek, S., and Salem, K. Management of partially safe buffers. *Computers, IEEE Transactions on 44*, 3 (1995), 394–407.

[31] Albrecht, C., Merchant, A., Stokely, M., Waliji, M., Labelle, F., Coehlo, N., Shi, X., and Schrock, C. E. Janus: Optimal flash provisioning for cloud storage workloads. In *Proceedings of USENIX ATC 2013*.

[32] Ananthanarayanan, G., Ghodsi, A., Warfield, A., Borthakur, D., Kandula, S., Shenker, S., and Stoica, I. PACMan: Coordinated memory caching for parallel jobs. In *Proceedings of USENIX NSDI 2012*.

[33] Anwar, A., Cheng, Y., and Butt, A. R. Towards managing variability in the cloud. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)* (May 2016), pp. 1081–1084.

[34] Anwar, A., Cheng, Y., Gupta, A., and Butt, A. R. Taming the cloud object storage with mos. In *Proceedings of the 10th Parallel Data Storage Workshop* (New York, NY, USA, 2015), PDSW '15, ACM, pp. 7–12.

[35] Anwar, A., Cheng, Y., Gupta, A., and Butt, A. R. Mos: Workload-aware elasticity for cloud object stores. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing* (New York, NY, USA, 2016), HPDC '16, ACM, pp. 177–188.

[36] Anwar, A., Cheng, Y., Huang, H., and Butt, A. R. Clusteron: Building highly configurable and reusable clustered data services using simple data nodes. In *8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 16)* (Denver, CO, 2016), USENIX Association.

[37] Arpaci-Dusseau, R. H., and Arpaci-Dusseau, A. C. *Operating Systems: Three Easy Pieces*, 0.91 ed. Arpaci-Dusseau Books, May 2015.

[38] Baek, S., Choi, J., Lee, D., and Noh, S. H. Energy-efficient and high-performance software architecture for storage class memory. *ACM Transactions on Embedded Computing Systems (TECS) 12*, 3 (2013), 81.

[39] Bailey, K. A., Hornyack, P., Ceze, L., Gribble, S. D., and Levy, H. M. Exploring storage class memory with key value stores. In *Proceedings of the 1st Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads* (2013), ACM, p. 4.

[40] Butt, A. R., Gniady, C., and Hu, Y. C. The performance impact of kernel prefetching on buffer cache replacement algorithms. *Computers, IEEE Transactions on 56*, 7 (2007), 889–908.

[41] Chen, Y., Alspaugh, S., and Katz, R. Interactive analytical processing in Big Data systems: A cross-industry study of MapReduce workloads. *PVLDB 5*, 12 (Aug. 2012), 1802–1813.

[42] Cheng, Y., Douglis, F., Shilane, P., Wallace, G., Desnoyers, P., and Li, K. Erasing belady's limitations: In search of flash cache offline optimality. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)* (Denver, CO, 2016), USENIX Association, pp. 379–392.

[43] Cheng, Y., Gupta, A., and Butt, A. R. An in-memory object caching framework with adaptive load balancing. In *Proceedings of the Tenth European Conference on Computer Systems* (New York, NY, USA, 2015), EuroSys '15, ACM, pp. 4:1–4:16.

[44] Cheng, Y., Gupta, A., Povzner, A., and Butt, A. R. High performance in-memory caching through flexible fine-grained services. In *Proceedings of the 4th Annual Symposium on Cloud Computing* (New York, NY, USA, 2013), SOCC '13, ACM, pp. 56:1–56:2.

[45] Cheng, Y., Iqbal, M. S., Gupta, A., and Butt, A. R. Cast: Tiering storage for data analytics in the cloud. In *ACM HPDC* (2015).

[46] Cheng, Y., Iqbal, M. S., Gupta, A., and Butt, A. R. Pricing games for hybrid object stores in the cloud: Provider vs. tenant. In *USENIX HotCloud* (2015).

[47] Cheng, Y., Iqbal, M. S., Gupta, A., and Butt, A. R. Provider versus tenant pricing games for hybrid object stores in the cloud. *IEEE Internet Computing 20*, 3 (2016), 28–35.

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

[49] Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., and Sears, R. Benchmarking cloud serving systems with YCSB. In *Proceedings of ACM SoCC 2010*.

[50] Dulloor, S. R., Kumar, S., Keshavamurthy, A., Lantz, P., Reddy, D., Sankaran, R., and Jackson, J. System software for persistent memory. In *Proceedings of the Ninth European Conference on Computer Systems* (2014), ACM, p. 15.

[51] Elghandour, I., and Aboulnaga, A. ReStore: Reusing results of MapReduce jobs. *PVLDB 5*, 6 (Feb. 2012), 586–597.

[52] Fan, Z., Du, D. H., and Voigt, D. H-arc: A non-volatile memory based cache policy for solid state drives. In *Mass Storage Systems and Technologies (MSST), 2014 30th Symposium on* (2014), IEEE, pp. 1–11.

[53] Ferreira, A. P., Zhou, M., Bock, S., Childers, B., Melhem, R., and Mossé, D. Increasing pcm main memory lifetime. In *Proceedings of the conference on design, automation and test in Europe* (2010), European Design and Automation Association, pp. 914–919.

[54] Gill, B. S. On multi-level exclusive caching: Offline optimality and why promotions are better than demotions. In *USENIX FAST* (2008).

[55] Grupp, L. M., Davis, J. D., and Swanson, S. The bleak future of nand flash memory. In *USENIX FAST* (2012).

[56] Guerra, J., Pucha, H., Glider, J., Belluomini, W., and Rangaswami, R. Cost effective storage using extent based dynamic tiering. In *Proceedings of USENIX FAST 2011*.

[57] Harter, T., Borthakur, D., Dong, S., Aiyer, A., Tang, L., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Analysis of HDFS under HBase: A Facebook Messages case study. In *Proceedings of USENIX FAST 2014*.

[58] Huang, S., Wei, Q., Chen, J., Chen, C., and Feng, D. Improving flash-based disk cache with lazy adaptive replacement. In *Mass Storage Systems and Technologies (MSST), 2013 IEEE 29th Symposium on* (2013), IEEE, pp. 1–10.

[59] Isard, M., Budiu, M., Yu, Y., Birrell, A., and Fetterly, D. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of ACM EuroSys 2007*.

[60] Jalaparti, V., Ballani, H., Costa, P., Karagiannis, T., and Rowstron, A. Bridging the tenant-provider gap in cloud services. In *Proceedings of ACM SoCC 2012*.

[61] Johnson, B. G., and Dennison, C. H. Phase change memory, Sept. 14 2004. US Patent 6,791,102.

[62] Jose, J., Banikazemi, M., Belluomini, W., Murthy, C., and Panda, D. K. Metadata persistence using storage class memory: Experiences with flash-backed dram. In *Proceedings of the 1st Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads* (2013), INFLOW '13.

[63] Jung, H., Shim, H., Park, S., Kang, S., and Cha, J. Lru-wsr: integration of lru and writes sequence reordering for flash memory. *Consumer Electronics, IEEE Transactions on 54*, 3 (August 2008).

[64] Kgil, T., and Mudge, T. Flashcache: A nand flash memory file cache for low power web servers. CASES, ACM.

[65] Kim, H., Seshadri, S., Dickey, C. L., and Chiu, L. Evaluating Phase Change Memory for enterprise storage systems: A study of caching and tiering approaches. In *Proceedings of USENIX FAST 2014*.

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

[67] Kreyszig, E. *Advanced Engineering Mathematics*, 10th ed. Wiley, August 2011.

[68] Krish K.R., Anwar, A., and Butt, A. R. hatS: A heterogeneity-aware tiered storage for Hadoop. In *Proceedings of IEEE/ACM CCGrid 2014*.

[69] Lee, E., Bahn, H., and Noh, S. H. Unioning of the buffer cache and journaling layers with non-volatile memory. In *Presented as part of the 11th USENIX Conference on File and Storage Technologies (FAST 13)* (2013), pp. 73–80.

[70] Lee, M., Kang, D. H., Kim, J., and Eom, Y. I. M-clock: migration-optimized page replacement algorithm for hybrid dram and pcm memory architecture. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing* (2015), ACM, pp. 2001–2006.

[71] Lee, S., Bahn, H., and Noh, S. H. Clock-dwf: A write-history-aware page replacement algorithm for hybrid pcm and dram memory architectures. *Computers, IEEE Transactions on 63*, 9 (2014), 2187–2200.

[72] Li, S., Hu, S., Wang, S., Su, L., Abdelzaher, T., Gupta, I., and Pace, R. WOHA: Deadline-aware Map-Reduce workflow scheduling framework over Hadoop clusters. In *Proceedings of IEEE ICDCS 2014*.

[73] Li, Z., Mukker, A., and Zadok, E. On the importance of evaluating storage systems' $costs. In *Proceedings of USENIX HotStorage 2014*.

[74] Lim, H., Fan, B., Andersen, D. G., and Kaminsky, M. Silt: A memory-efficient, high-performance key-value store. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (2011), ACM, pp. 1–13.

[75] Lim, H., Herodotou, H., and Babu, S. Stubby: A transformation-based optimizer for MapReduce workflows. *PVLDB 5*, 11 (July 2012), 1196–1207.

[76] Liu, Z., Liu, I., Low, S., and Wierman, A. Pricing data center demand response. *SIGMETRICS Perform. Eval. Rev. 42*, 1 (June 2014), 111–123.

[77] Londoño, J., Bestavros, A., and Teng, S.-H. Colocation games: And their application to distributed resource management. In *USENIX HotCloud* (2009).

[78] Lu, Y., Shu, J., and Zheng, W. Extending the lifetime of flash-based storage through reducing write amplification from file systems. In *USENIX FAST* (2013).

[79] Mao, M., and Humphrey, M. Auto-scaling to minimize cost and meet application deadlines in cloud workflows. In *Proceedings of ACM/IEEE SC 2011*.

[80] McDougall, R., and Mauro, J. Filebench. *URL: http://www. nfsv4bat. org/Documents/nasconf/2004/filebench. pdf (Cited on page 56.)* (2005).

[81] Megiddo, N., and Modha, D. S. Arc: A self-tuning, low overhead replacement cache. In *FAST* (2003), vol. 3, pp. 115–130.

[82] Mihailescu, M., Soundararajan, G., and Amza, C. MixApart: Decoupled analytics for shared storage systems. In *Proceedings of USENIX FAST 2013*.

[83] Narayanan, D., Donnelly, A., and Rowstron, A. Write off-loading: Practical power management for enterprise storage. In *Proceedings of 6th USENIX Conference on File and Storage Technologies (FAST âĂŹ08)* (San Jose, CA, February 2008), USENIX.

[84] Narayanan, D., Thereska, E., Donnelly, A., Elnikety, S., and Rowstron, A. Migrating enterprise storage to ssds: analysis of tradeoffs. In *Proceedings of EuroSys 2009* (Nuremberg, Germany, March 2009), ACM.

[85] Park, S.-y., Jung, D., Kang, J.-u., Kim, J.-s., and Lee, J. Cflru: a replacement algorithm for flash memory. In *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems* (2006), ACM, pp. 234–241.

[86] Puttaswamy, K. P., Nandagopal, T., and Kodialam, M. Frugal storage for cloud file systems. In *Proceedings of ACM EuroSys 2012*, ACM.

[87] Qureshi, M. K., Srinivasan, V., and Rivers, J. A. Scalable high performance main memory system using phase-change memory technology. *ACM SIGARCH Computer Architecture News 37*, 3 (2009), 24–33.

[88] Santana, R., Lyons, S., Koller, R., Rangaswami, R., and Liu, J. To arc or not to arc. In *7th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 15)* (2015).

[89] Sen, S., Joe-Wong, C., Ha, S., and Chiang, M. Incentivizing time-shifting of data: a survey of time-dependent pricing for internet access. *IEEE Communications Magazine 50*, 11 (November 2012), 91–99.

[90] Shi, W., Zhang, L., Wu, C., Li, Z., and Lau, F. C. An online auction framework for dynamic resource provisioning in cloud computing. In *ACM SIGMETRICS* (2014).

[91] Shvachko, K., Kuang, H., Radia, S., and Chansler, R. The Hadoop Distributed File System. In *Proceedings of IEEE MSST 2010*.

[92] Smullen, C. W., Mohan, V., Nigam, A., Gurumurthi, S., and Stan, M. R. Relaxing non-volatility for fast and energy-efficient stt-ram caches. In *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on* (2011), IEEE, pp. 50–61.

[93] Strukov, D. B., Snider, G. S., Stewart, D. R., and Williams, R. S. The missing memristor found. *nature 453*, 7191 (2008), 80–83.

[94] Wang, H., and Varman, P. Balancing fairness and efficiency in tiered storage systems with bottleneck-aware allocation. In *Proceedings of USENIX FAST 2014*.

[95] Wieder, A., Bhatotia, P., Post, A., and Rodrigues, R. Orchestrating the deployment of computations in the cloud with Conductor. In *Proceedings of USENIX NSDI 2012*.

[96] Wong, T. M., and Wilkes, J. My cache or yours? making storage more exclusive. In *Proceedings of the General Track of the Annual Conference on USENIX Annual Technical Conference* (2002), USENIX '02.

[97] Wu, G., and He, X. Delta-FTL: Improving ssd lifetime via exploiting content locality. In *ACM EuroSys* (2012).

[98] Yang, J., Wei, Q., Chen, C., Wang, C., Yong, K. L., and He, B. Nv-tree: Reducing consistency cost for nvm-based single level systems. In *13th USENIX Conference on File and Storage Technologies (FAST 15)* (2015), pp. 167–181.

[99] Yoo, S., Lee, E., and Bahn, H. Ldf-clock: The least-dirty-first clock replacement policy for pcm-based swap devices. *JOURNAL OF SEMICONDUCTOR TECHNOLOGY AND SCIENCE 15*, 1 (2015), 68–76.

[100] Yuan, D., Yang, Y., Liu, X., and Chen, J. A data placement strategy in scientific cloud workflows. *Future Generation Computer Systems 26*, 8 (2010), 1200 – 1214.

[101] Zaharia, M., Borthakur, D., Sen Sarma, J., Elmeleegy, K., Shenker, S., and Stoica, I. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of ACM EuroSys 2010*.

[102] Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M. J., Shenker, S., and Stoica, I. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation* (2012), USENIX Association, pp. 2–2.

[103] Zhou, P., Zhao, B., Yang, J., and Zhang, Y. A durable and energy efficient main memory using phase change memory technology. In *ACM SIGARCH computer architecture news* (2009), vol. 37, ACM, pp. 14–23.

[104] Zhou, Y., Philbin, J., and Li, K. The multi-queue replacement algorithm for second level buffer caches. In *USENIX Annual Technical Conference, General Track* (2001), pp. 91–104.