ISOLATION IN CLOUD STORAGE

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Ji-Yong Shin

January 2017

ISOLATION IN CLOUD STORAGE

Ji-Yong Shin, Ph.D.

Cornell University 2017

Due to the presence of large and heterogeneous user workloads and concurrent I/O requests, it is important to guarantee isolation in cloud storage systems. This dissertation explores isolation in cloud storage systems and makes fundamental contributions that advance the state of the art in supporting such isolation.

Specifically, this dissertation focuses on three key areas necessary for isolation in cloud storage systems: performance isolation, transactional isolation, and fine-grained consistency control. Regarding the first, performance isolation, resource contention in storage systems is often unavoidable under concurrent users and the contention allows a user to affect the performance experienced by other users. In particular, a single user can easily degrade the performance for all other users in a disk-based system because the disk performance is inherently susceptible to random I/O requests. Second, to maintain consistent data states under concurrent I/O requests, systems have implemented transactional isolation in high layers of the storage stack. However, different implementations in high layers of the storage stack make the support for transactional isolation redundant and transactions executed by different applications incompatible with each other. Thus, portable and compatible transactional isolation is required, as well as reconsideration of the layers of the storage stack in which transactional isolation should be placed. Finally, distributed systems often provide per-client views of the system by using client-centric consistency semantics to trade off

consistency and performance. While cloud storage servers have tens of parallel storage devices and CPU cores, which make the server comparable to a distributed system, the potential trade-off between consistency and performance within a server has never been explored.

We subsequently make three contributions embodied in approaches to addressing various isolation challenges. First, we present an approach that achieves performance isolation by resolving I/O contention using a chained-logging design. The chained-logging design retains at least one disk for sequentially logging without I/O contention even under garbage collection and systematically separates read and write operations to different disks. We implemented an instance of the approach in a system called Gecko. Second, we investigate an approach for block-level transactions that support portable and compatible transactional isolation. The block-level transaction facilitates transactional application designs in any layer of the storage stack and enables cross-application transactions. We implemented an instance of the approach in a system called Isotope. Finally, we define a new class of systems called StaleStore, which can trade off consistency and performance within a server using stale data, and we study the necessary functionality and interface to take advantage of this trade-off. Yogurt, an instance of StaleStore, explores different versions of data and estimates the access cost for each version under client-centric consistency semantics to trade off consistency and performance within a server. Together, these three approaches are important steps towards isolation in cloud storage systems.

BIOGRAPHICAL SKETCH

Ji-Yong Shin was born in Seoul, Republic of Korea, where he grew up dreaming of becoming a scientist or an engineer like his father. When he joined Yonsei University, Seoul, Republic of Korea in 2000, his interest in programming and the dot-com bubble influenced him to become a computer scientist.

During his college years, Ji-Yong loved systems and computer architecture courses and enjoyed programming. Two years after he studied computer science at his college, he joined the Republic of Korea Army to fulfill the mandatory military service for Korean men. He worked as a journalist and translator for two years. Although his position in the military had nothing to do with computer science, his military experience taught him the true meaning of endurance and perseverance, which later helped him to pursue the Ph.D. degree. Two years after he finished his military service, he earned the B.S. degree in Computer Science and Industrial Engineering with a minor in Electrical and Electronics Engineering.

To extend his horizons in computer science, Ji-Yong joined KAIST (Korea Advanced Institute of Science and Technology), Daejeon, Republic of Korea in 2007. He pursued his M.S. degree under the supervision of Prof. Seungryoul Maeng while focusing on designing new computer architectures and NAND flash-based SSDs. During his time at KAIST, he broadened his view by meeting with many smart colleagues and doing an internship at Microsoft Research Asia, Beijing, China. Then he decided to join a Ph.D. program in the U.S. to further extend his understanding of computer science.

While he was applying for Ph.D. programs in the U.S., he did internships at Microsoft Research, Redmond, WA and IBM T.J. Watson Research Center, Hawthorne, NY. The projects Ji-Yong participated in were related to systems, which were somewhat different from what he used to work on at KAIST. Such work experience and his growing interest in systems led him to pursue the Ph.D. degree in the systems field.

In 2010, Ji-Yong joined Cornell University's Ph.D. program. During the first two years, he jointly worked with Prof. Hakim Weatherspoon and Prof. Emin Gün Sirer on datacenter networks and then switched to working on storage systems under the supervision of Prof. Hakim Weatherspoon alone. Besides the research at Cornell, Ji-Yong worked at Microsoft Research Silicon Valley Center, Mountain View, CA and Google, Mountain View, CA as a summer intern. After spending six years at Cornell, Ji-Yong is now looking forward to joining Yale University as a postdoctoral associate.

For my family and friends

who always supported me, stood by me, and prayed for me.

ACKNOWLEDGEMENTS

First, I would like to thank the faculty members at Cornell. I would like to express my deepest gratitude to my advisor, Hakim Weatherspoon, who has patiently guided me and taught me how to become a systems researcher throughout my degree program. Through him, I could learn the insight to look into problems from various angles, how to collaborate with others, and effectively and passionately present research findings. He always inspired me and encouraged me when I was questioning about my research directions and outcomes, and stepped out to help not only regarding research but also regarding small issues surrounding the life of a Ph.D. student. Next, I would like to thank Emin Gün Sirer for showing me how to do hacking-based systems research and teaching me how to properly write papers during the collaboration with him. I am grateful to Andrew Myers and Edward Suh who gracefully accepted to become by dissertation committee members and provided me with valuable pieces of advice and comments for my research.

Second, I was lucky to work with many great research collaborators. I am very grateful to work with Mahesh Balakrishnan who gave constructive feedbacks and inspiring ideas all the time. I worked with Mahesh on many projects and he was like my second advisor. I could learn how to take care of small details of research projects and ways to write convincing papers from him. I thank Tudor Marian who has been a great consultant and mentor for difficult programming challenges including hacking Linux kernel. Tudor was also a great host during my internship at Google. I thank Lakshmi Ganesh for her collaboration on the Gecko project and Jakub Szefer for opening the door to integrating my research work into hardware. I appreciate Darko Kirovski, Rama Ramasubramanian, and Chandu Thekkath for being wonderful mentors during my internships at Microsoft Research and letting me know and experience how research can impact real life.

Third, I thank my colleagues in the Computer Science department and members of the Systems Lab who have been great company, especially Ki-Suh Lee, Han Wang, Zhiming Shen, Qin Jia, Robert Escriva, Deniz Altinbuken, Elisavet Kozyri, Zhiyuan Teo, Stavros Nikolaou, Hussam Abu-Libdeh, Dan Williams, Joonsuk (Jon) Park, Moontae Lee, and Jaeyong Sung.

Forth, I would like to thank the people outside of the department. I thank Younghwa Seok for always supporting and encouraging me especially during several challenging moments of the degree program. I also thank Younghwa for helping proofread many chapters of this dissertation. I thank Jangwoo Kim, Heyjun Park, Young-Hye Song, Jaesun Lee, Daniel Lee, Goeun Lee, and members of the Korean Church at Cornell and the Cornell Korean Tennis Club for being wonderful friends who made the time I spent in Ithaca and Cornell beautiful and enjoyable.

Finally, I would like to thank my family, father, mother and brother, for their endless love, support, encouragement and prayer. Without them, I would not have made it this far.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Individuals and broad cross-sections of organizations — including tech services, financial services, education, media and publishing, hardware industries, business services, software services, and others — rely heavily on the cloud [105]. *Cloud computing* refers to both applications delivered as services over the Internet and the system software and hardware in the datacenters that provide those services, whereas the *cloud* itself refers to the hardware and software for cloud computing [31]. *Cloud storage* is a part of cloud computing services that supports the storage functionality. The extensive use of the cloud has resulted in individuals, enterprises, and governments storing a vast amount of data in cloud storage systems. *Cloud providers* report that they host trillions of data objects and handle millions of concurrent requests [22, 23, 3]. Further, the use of the cloud and cloud storage is expected to grow [16]. The challenge is orchestrating and ensuring individual user performance and their requests as if each user were isolated from massive numbers of users especially with regard to cloud storage systems.

Guaranteeing isolation is crucial to handling high volumes of concurrent requests in cloud storage systems, but it entails many challenges. Isolation refers to encapsulating users or processes in an independent execution environment or ensuring users are less affected or not at all affected by others [63, 134]. For example, isolation of performance in storage systems prevents one user from affecting the performance for other users. Another aspect of isolation is found in data access semantics — definitions, restrictions and formal rules that govern how data is accessed [88]. Data access semantics enable multiple users to simultaneously access shared data without encountering any anomalous data states by maintaining isolated data access rules [38]. However, high volumes of concurrent requests, a mix of user workloads, heterogeneous and parallel hardware configurations, and so on within cloud storage systems provide new challenges and opportunities to achieve isolation.

Therefore, this dissertation focuses on addressing the following research question: *how can a cloud storage system achieve isolation?* We explore this question in three key areas: performance isolation, transactional isolation, and fine-grained consistency control. This chapter first presents background information regarding the research question and then elaborates on the challenges in each key area. Based on the challenges, we present detailed research questions and summarize our contributions.

1.1 Cloud Storage Servers

We generalize the definition of a cloud storage system as any storage system that is shared and accessed concurrently by multiple users, processes, or threads. It varies in scale ranging from a single block device to large-scale key-value stores and filesystems to a large (distributed) system that consists of thousands of servers [1, 2, 8, 14]. In this dissertation, we focus on key characteristics: shared and concurrently accessed.

We define a single cloud storage server as an individual server that is part of a cloud storage system. The cloud storage server also shares the same key characteristics of being shared and concurrently accessed by multiple users. A cloud storage server can serve the user by itself and can be used as a basic building block to construct a large-scale storage system. This dissertation focuses on a cloud storage server.

1.1.1 Cloud Storage Server Trends

We compare the server specifications from years 2000, 2006, and 2016, to understand how the storage server has evolved. Table 1.1 summarizes the server trends, which reflect how the hardware technology for servers has evolved. Dell PowerEdge models 2450 [49], 2850 [50], and R930 [51] from years 2000, 2006, and 2016, respectively, are compared by showing the CPU types and the total number of cores, last level cache and memory sizes, network interfaces, storage drives, and external ports. The servers are equipped with the latest available hardware when they were manufactured and are specialized for storage services.

The "CPU" row of the table indicates that the total number of cores has increased from 2 and 4 to 96 in the year 2016. The number is increased by 48X and 24X compared to years 2000 and 2006, respectively. The increase reflects the CPU development trend of increasing number of cores [25]. Assuming that a core hosts one user at a time, the number of parallel accesses in a cloud storage server has increased by almost 50 times compared to the year 2000. The increased number of CPU cores and parallelism emphasize the importance of isolation in cloud storage servers.

Similarly, the last level cache capacity has grown by 30X (from 2MB to 60MB), the memory size increased by 384X to 3000X (from 2GB and 16GB to 6TB), and the total bandwidth – the amount of data that can be transferred per

Year	2000	2006	2016
Model	Dell PowerEdge 2450	Dell PowerEdge 2850	Dell PowerEdge R930
CPU (total # of cores)	2 × single core Intel Pentium III Xeon @ 600 MHz (2)	2 × dual core Intel Xeon @ 2.8GHz (4)	4 × 24 core Intel Xeon @ 2.1GHz (96)
Last level cache	None (CPU internal)	2MB	60MB
Memory	2GB	16GB	6TB
Network interfaces	1 × 100Mbps Ethernet	2 × 1Gbps Ethernet	2 × 1Gpbs Ethernet, 2 × 10Gbps Ethernet
Storage drives	4 × SCSI HDD, 1 × tape drive	6 × SCSI HDD (one can be used for a tape drive)	24 × SAS HDD or SAS/SATA SSD, 8 × PCIe SSD
External ports	1 × serial, 1 × parallel	2 × PCIe, 1 × PCI-X	10 × PCIe

Table 1.1: Dell server specifications from year 2000 [49], 2006 [50], and 2016 [51].

second – of network interfaces has expanded by 11X to 220X (from 100Mpbs and 2Gpbs to 220Gbps) during one and a half decades. Together, the CPU, the last level cache, memory, and network interfaces show that a single cloud storage server in 2016 is as powerful as tens to hundreds of servers in years 2000 and 2006. The hardware resource trends in the table imply that the cloud storage server has become powerful and offers great parallelism.

The "storage drives" row in the table highlights that the servers rely on hard disk drives (HDD). A HDD is composed of multiple platters, a spindle, and an arm [122]. The platter is a round magnetic disk, where the data is stored. Platters have circular tracks, which form concentric circles from the center to the edge. Each track consists of sectors that are data accessing units of a HDD. A sector has a fixed capacity such as 512 bytes, but the physical size of a sector

Year	1997	2007	2016	2016
Model	Western Digital Caviar AC22500	Western Digital WD1600AABS	HGST Ultrastar He10	HGST Ultrastar C10K1800
Capacity	2.5 GB	250GB	10 TB	1.2TB
Rotational speed	5400RPM	7200RPM	7200RPM	10520RPM
Average Latency	5.8ms	4.2ms	4.2ms	2.8ms
Average seek time	11.5ms	8.9ms	8.3ms	3.3ms
Interface (bandwidth)	EIDE (264Mb/s)	SATA II (3Gb/s)	SATA III (6Gb/s)	SAS-3 (12Gb/s)
Form factor (platter size)	3.5in	3.5in	3.5in	2.5in

Table 1.2: Hard disk drive specifications from year 1997 [140], 2007 [139], and 2016 [70, 69].

varies as the length of a track varies. A spindle holds the platters together in a cylindrical shape and rotates the platters. The arm has a head at its tip and the head accesses data from the platter. The arm mechanically moves between the inner and the outer tracks of platters. The movement of the arm and the rotation of platters enable the head to access the entire surface of the platter. To access a sector, the disk arm moves to find the track of the sector on a platter, which is known as the seek operation, and the platter rotates so that the sector can be accessed by the head. When the head accesses data in sequential addresses, the data is read from the same track and the arm does not need to seek. On the other hand, when the head accesses random addresses, the arm seeks to find the track. The seek time dominates the HDD access latency, which is the time interval between a request and the following response. The problem is that the latency for the seek has not improved much for decades: the average seek time improved from approximately 12ms to 3ms over two decades [56].

Table 1.2 shows the HDD specifications from 1997 to 2016. HDDs with the largest capacity in each year are selected. The HDD capacity has grown by 4000X (from 2.5GB to 10TB), but the rotational speed and the average latency have improved by only 2X (from 5.4K RPM to 10.5K RPM and from 5.8ms to 2.8ms). The average latency is mainly limited by the seek time that hardly decreases. As shown in Table 1.2, the seek time is noticeably reduced only when using a smaller form factor: smaller platter size makes the seek distance shorter. The gap of improvements found in the HDD versus CPU and memory technologies makes storage operations relatively slower.

Despite the slow performance (latency) trends, the number of HDDs in servers has increased by 5X to 8X (from 4 and 6 to 32) during the past one and a half decades and the bandwidth to storage devices has become greater ("Storage drives" row of Table 1.1 and "Interface" row of Table 1.2). The bandwidth is capped by the communication interfaces between the server and the storage device and Table 1.3 summarizes the interfaces. The storage interface has evolved from Enhanced Integrated Device Electronics (EIDE) to Serial Attached Small Computer System Interface (SAS) and the bandwidth has increased steadily for two decades. Serial, Parallel, Peripheral Component Interconnect eXtended (PCI-X), and Peripheral Component Interconnect Express (PCI-e) were mainly used to connect non-storage devices to the server, but PCIe is being actively used for high-end solid state drives (SSD).

NAND-flash-based SSDs are persistent storage devices, which are being widely deployed in cloud storage servers as shown in the "storage drives" row of Table 1.1. A SSD is composed of multiple NAND flash memory chips. The NAND flash memory stores data persistently using a floating gate, which is an

Name	Bandwidth	Common Usage
EIDE (Enhanced Integrated Drive Electronics)	16.67Mbps	HDD, CD/DVD Drive
SCSI (Small Computer System Interface)	160Mbps - 640Mbps	HDD, CD/DVD Drive
SATA (Serial AT Attachment)	1.5Gpbs - 16Gbps	HDD, SSD, CD/DVD Drive
SAS (Serial Attached SCSI)	3Gbps - 22.5Gbps	HDD, SSD
Serial	110bps - 256Kbps	Terminal, printer, phone, mouse
Parallel	150Kbps	Zip drive, scanner, modem, external HDD
PCI-X (Peripheral Component Interconnect eXtended)	6.4Gbps - 34.4Gpbs	Network interface card, graphics card
PCIe (Peripheral Component Interconnect Express)	2Gbps - 252Gbps	SSD, Network interface card, graphics card

Table 1.3: Communication ports and interfaces [89].

electronically controlled storage element. Thus, the NAND flash memory does not have any mechanical moving parts and as a result can access random data locations with much lower latency compared to a HDD. Thus, the data access speed of a SSD is generally faster than that of a HDD [39]. The data access latency of a SSD is bounded by the NAND flash memory speed, which varies depending on the vendor. The flash memory requires erase operations before writing data to the same location. Typically, the read, write, and erase latencies are approximately 20μs, 300μs, and 2ms, respectively [39, 58]. The bandwidth of an SSD can increase by utilizing multiple NAND flash memory chips in parallel and employing wider storage interfaces. Due to the high cost, however, SSDs are used for special purposes rather than completely replacing HDDs [91].

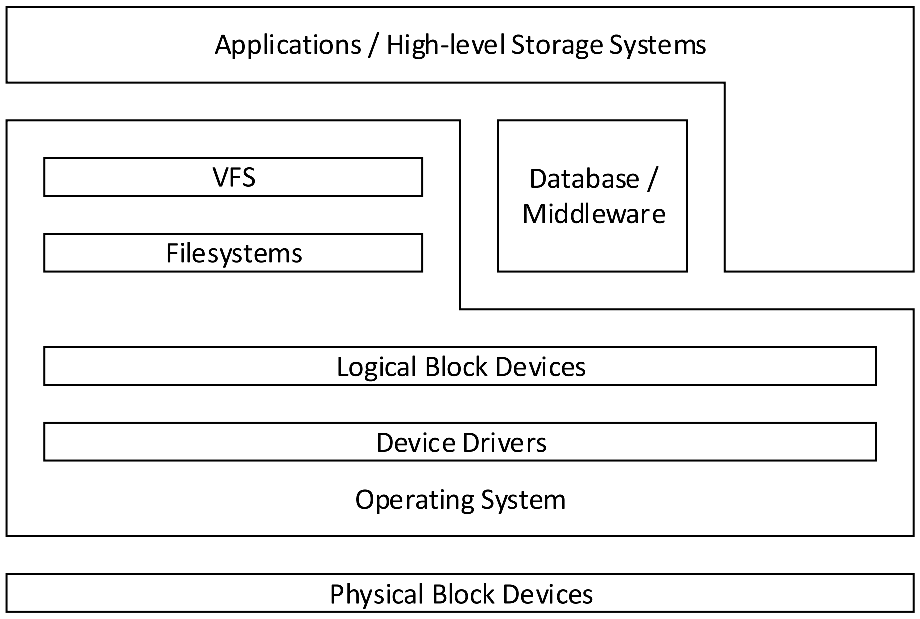


Figure 1.1: Linux storage stack.

1.1.2 Cloud Storage Stacks

Understanding the storage stack helps understand the cloud storage server as the cloud systems and applications run in different layers of the stack. We detail the storage stack based on a Linux operating system [40]. A Linux operating system has physical block devices, device drivers, logical block devices, filesystems, virtual filesystems (VFS), and applications in the storage stack (Figure 1.1). The filesystem and virtual filesystem layers can be replaced with databases and middleware layers depending on the needs of applications. Device drivers interface between operating system and physical block devices to send and receive hardware-specific commands. Logical block devices are software abstractions of physical block devices. Filesystems use the block interface to implement file abstractions, and the virtual filesystem provides a uniform interface to access files on different filesystems. Databases store data in row and column formats for easy composition and searching, and middleware interfaces between applications and the operating system.

A cloud storage application typically runs on filesystems and databases, but it can bypass them and directly access logical block devices. High-level storage systems, such as library databases and even distributed key-value stores, are often built in the application layers. Similar to the applications, high-level storage systems can access filesystems, databases, and logical block devices directly depending on their designs.

Cloud services often use virtualization to multiplex and share physical hardware machines [34]. Virtualization is enabled by hypervisor software, which creates and manages multiple virtual machines (VM) on a physical machine. VMs are emulated computer machines that can be accessed independently. VMs have virtualized hardware devices, such as virtual CPU, virtual memory, virtual disk, and so on, which are created by the hypervisor. Based on the virtualized hardware devices, VMs can run software as if the software is running on a physical machine. Hypervisors run directly on a physical hardware machine (type I hypervisor) or inside the application layer of an operating system (type II hypervisor) [99]. The type I hypervisor runs on the same layer as the operating system, so the virtualization overhead is relatively lower than that of the type II hypervisor.

1.2 Isolation

In this dissertation, we focus on three different topics regarding isolation: performance isolation, transactional isolation, and client-centric consistencies. The following subsections provide the definition and the context for each topic.

1.2.1 Performance Isolation

Performance isolation is a property that minimizes noticeable contention of resources and access time delays in systems to make users unaware of each other's behaviors [134, 63]. Performance isolation is not meant to prevent sharing of the physical storage – for example, share part of the storage bandwidth – but rather filtering out side effects of a user that significantly slow down others. For example, IceFS provides performance isolation to each user by flushing data of each user independently from memory to disk [80]. On the other hand, ext3 filesystem, a filesystem commonly used in Linux, flushes all data in memory to disk even when only one user calls *sync*. Thus, other users that did not call *sync* can be significantly slowed down because their data is unnecessarily flushed [80].

Performance isolation is indispensable in storage systems. Storage bandwidth in systems is typically lower than the CPU and memory bandwidth and accessing the storage has been a long-standing bottleneck. Hence, when multiple users issue requests at a high rate, the contention for the storage access becomes noticeable. The access latency to the storage is also very high compared to that of CPU or memory, so queued input/output (I/O) requests under concurrent storage accesses can cause significant delays. To provide performance isolation and hide resource contention, for example, cloud providers predict the storage bandwidth conservatively, limit the maximum storage access bandwidth of a user to be a very small fraction of the total bandwidth, and give vague or no guarantees for the latency [35].

1.2.2 Transactional Isolation

Transactional isolation is a property that defines how concurrent transactions should access data independently without violating the integrity of data in storage systems. A transaction is a sequence of operations carried out in a reliable, independent and consistent way on a shared storage [37]. It is a programming model that has been used in databases for decades. Transactions have four properties called ACID: atomicity, consistency, isolation, and durability (see Appendix A). Atomicity guarantees that a transaction is executed completely or not at all. Consistency guarantees a transaction changes the storage state from one consistent state to another consistent state, which does not violate the integrity constraint – predefined rules for how and in which format the data should be stored [127]. Isolation guarantees that concurrent transactions are executed in an order that does not violate consistency. Finally, durability ensures that data updates successfully made by a transaction are stored durably and cannot be lost.

While consistency defines how the data should transform in each step in a storage system, transactional isolation defines how multiple users should access data concurrently without interfering with others and not violating consistency; it defines the ordering constraints of multiple transactions accessing shared data. Keeping consistency is straightforward under a single user, but concurrency complicates maintaining consistency. When multiple users access a storage system simultaneously, for example, data pieces that should be modified together can be updated partially or at different times, which can lead to violating consistency. Thus, transactional isolation should coordinate the data access from multiple transactions to keep the storage consistent. As with performance isolation, transactional isolation is important especially as concurrency increases in cloud storage systems.

There are several transactional isolation semantics [37], but we discuss two: strict serializability and snapshot isolation. Strict serializability is equivalent to scheduling concurrent transactions sequentially one after another with no overlapping transactions while preserving the order observed by the transaction issuing processes [37]. Strict serializability is the strongest guarantee, which leads to the same result as transactions executing one at a time. On the other hand, snapshot isolation is a weaker guarantee than strict serializability because snapshot isolation allows interleaved transactions that are prohibited by strict serializability. Snapshot isolation is a guarantee that all reads by a transaction see all updates by transactions that have successfully completed before the transaction started [36]. A snapshot refers to a state of the storage system at a particular time point. Namely, the reads of the transaction are served from a consistent snapshot of the storage system that was taken at the beginning of the transaction. Still, snapshot isolation is widely used in databases for performance reasons; it allows for greater concurrency (overlap) in transaction executions.

There are two different implementations of the transactional isolation semantics known as pessimistic concurrency control and optimistic concurrency control [37]. Pessimistic concurrency control assumes that there are always transactions that have conflicting data accesses. It uses locks to prevent transactions from executing prohibited data accesses. Once a transaction locks data, no other transactions can access the data until the same transaction unlocks the data. On the other hand, optimistic concurrency control assumes that there are no transactions that have conflicting data accesses. It lets any transactions access any data. However, the updates made by a transaction are not directly applied to the storage system. At the end of a transaction execution, the transaction is tested whether it has any conflicting data accesses with other transactions. If there is no conflict, the transaction commits, which means the transaction succeeded and its updates are applied to the storage. If a conflict is found, only one of the conflicting transactions commits and all others abort, which means the transaction failed and the updates are not applied to the storage. The pessimistic approach conservatively blocks the execution of transactions while the optimistic approach allows transactions to continue. Thus, if conflicts are rare, the optimistic approach generally performs better. However, if there are many conflicting transactions, the optimistic approach can suffer from a huge amount of aborts [37].

1.2.3 Client-Centric Consistencies

A client-centric consistency is a class of weak consistency semantics in distributed systems, which only defines per-client guarantees. Distributed systems often replicate data across different servers and assume that the updates are propagated from one server to another slowly with an uncertain amount of network delays [133, 78]. Consistency semantics define how and in which order an update to a data object is propagated to servers and how users access the data [128]. Consistency semantics take into account the propagation delay of updates and different versions of data that exist in the servers. Data-centric consistency semantics enforce a consistent view of the entire storage system by ordering sequences of data accesses similar to transactional isolation semantics. Client-centric consistency semantics provide isolated views of the storage system to each user with guarantee rules per data object rather than a sequence of data accesses in the storage system (see Appendix B).

Client-centric consistencies focus on the consistent view of a storage system centered from a client and do not guarantee anything regarding concurrent data accesses among different clients. The followings are examples of client-centric consistency guarantees:

- *Bounded staleness* guarantees that the data read by a client once was the latest data within a time bound. The time bound can be replaced with the number of updates.
- *Monotonic reads* guarantee that the value of a data read by a client is the same or newer than the previously read value of the same data by the same client.
- *Read-my-writes* guarantee that the value of a data read by a client is the same or newer than the previously written value of the same data by the same client.

The following example describes how client-centric consistency works. Assume that there are two users A and B, two servers S_1, and S_2, and two data objects X, and Y, in a distributed storage system. The data values for each object are represented with version numbers, for example, x_1 is the first version of data X's value. User A reads values x_5 of X, adds value one to x_5, and writes the result as y_5 to data Y in server S_1. After a while, due to an unstable network condition, server S_1 holds values x_5 and y_5, but S_2 holds values x_4 and y_5. Then, user B, who is closer to S_2 reads data Y from S_2 and gets y_5. User B tries to read data X next. Under a data-centric consistency model, the value y_5 is dependent on x_5, because x_5 is used to generate y_5, so user B should read x_5 of data X from server S_1 which can take a long time. Under a client-centric consistency model, say monotonic-reads, the dependency created by user A has nothing to do with user B. Thus, as long as user B did not read value x_5 before, user B can access the closer server S_2 to quickly read the value x_4 of X.

Under client-centric consistency semantics, the general assumption is that a server returns only one version of data (locally latest to the server) at any given time and the returned version of the data can differ only among different servers. Later in the dissertation, we break this assumption and return multiple versions of data from a server.

1.3 Challenges

In this section, we introduce the details of the problems and challenges for supporting isolation in cloud servers and explain three research questions that this dissertation addresses.

1.3.1 Lack of Cloud Storage Server Performance Isolation

Under concurrent accesses, the performance of a cloud storage server is difficult to predict unless the storage workload is well known and does not change over time [136]. The underlying assumption of the cloud is that a storage workload can run anywhere, so a workload of a user can be co-located on a same physical server with workloads run by others [148, 121]. If the user gets lucky, the workload is placed on an idle server or a server with well behaving (e.g. sequential disk) workloads and runs under good performance. If not, the workload is located on a busy server or with misbehaving (e.g. random disk) workloads and suffers from a bad performance. In the latter case, the main causes for the bad performance are often found from the use of hard disk drives and the lack of performance isolation.

The fundamental challenge is overcoming the mechanical characteristics of the disk that make one workload susceptible to another while supporting increased user parallelism. Disks have been notorious for being poor at handling random I/Os because of how the mechanical parts are designed [96]. Random I/Os make the disk arm seek, which significantly delays the I/O. In a multi-user environment that shares a disk, this can be especially harmful: when there is a user issuing random I/Os, all other users suffer from random seek operations. Even if there is no user issuing random I/Os, sequential I/O requests from multiple users can be mixed together to behave like random I/Os. Namely, the performance of a disk is very easily affected by the characteristics of workloads and is worsened even under well-behaving workloads if they run together. Importantly, the performance becomes far worse than dividing the maximum disk performance by the number of concurrent users when random I/Os are present. Therefore, a disk is inherently bad for performance isolation. Although the disk characteristics remain the same and the performance has not been improved as much as other hardware, a cloud storage server holds tens of disks and mix of other storage devices such as SSDs. Under the cloud storage server environment, we investigate the first research question: *given the diversity and abundance of storage devices, how can we achieve performance isolation on a disk-based storage system?*

A log-structured filesystem (LFS) [107] concatenates all data that is being written to make write operations sequential. Sequential writes significantly improve the performance of the disk-based storage system. However, LFS needs to recycle disk blocks which have been overwritten using an operation called garbage collection. Garbage collection involves reading old data blocks and relocating only the data blocks that are up-to-date. In LFS, the garbage collection and read operations of applications cause random I/Os in disks and degrade the performance. SSDs have been proposed as caches on top of disks to prevent random seek operations [106], but the cache misses still have a significant performance impact. Moreover, limited lifetime, garbage collection of flash pages, and the high cost of SSDs make it less suitable to cache all I/Os.

Disks are excellent at handling sequential I/Os, but very bad at handling random I/Os. The performance for handling sequential and random I/O differs by orders of magnitude [118]. Thus, a disk that always accesses data sequentially can outperform multiple disks that access data randomly. Similarly, reserving a disk to always access data sequentially can lead to better performance isolation, as the random access to disk is disruptive for performance isolation. Therefore, we investigate keeping at least one disk to always access data sequentially, when multiple disks are present. As an instance of this approach, we explore chained logging design, which uses both logging and SSD-based caching to always keep one disk to write sequentially without contention. Gecko is an instance of chained logging and uses the log-structured design that eliminates read-write contention by chaining together a small number of disks into a single log, effectively writing to one drive after another once a drive gets full. As a result, writes proceed sequentially to only one drive, which we call the tail drive, at any given moment. Garbage collection reads and application reads are restricted to non-tail drives and do not contend with writes with the help of a SSD caching policy specific to the tail drive. The tail drive always remains executing sequential I/O and achieves better performance and performance isolation compared to the state of the art. The details of chained logging and Gecko can be found in Chapter 3.

1.3.2 Lack of Cloud Storage Server Transactional Isolation

Cloud applications are developed with multiple concurrent users in mind. Transactional models, which include transactional isolation, have been used in databases for decades and are well adopted in the cloud for handling concurrency [38, 78]. Transactions have been implemented in many systems and applications in the high layers, such as filesystem and application layers, of the storage stack. Traditionally, the low layer of the storage stack, such as the logical block layer, have been used to handle simple read and write operations and rich functionalities such as transactions have not been implemented. Hence, applications and systems in the high layer are built without transactional functionalities support by the lower layer and implement transactions of their own. Implementing transactions repeatedly in every new application is a big burden for developers as the implementation is sophisticated and different implementations are typically not compatible with each other. Therefore, for the purpose of supporting transactions in cloud storage servers, where multiple systems and applications run together, implementing transactions at the application level is not sustainable.

The fundamental challenge is figuring out how to support transactions from the low layer of the storage stack so that systems and applications in the cloud environment do not have to implement transactions of their own and have a compatible transaction support with each other. The transaction implementation and API in the low layer of the stroage stack must be general and portable so that any storage or software stack can easily use. Thus, the API should be pushed down to the lowest common software stack, which is the logical block storage layer. The support for transactions, especially isolation, has not been implemented in the block layer not only because the block layer has been traditionally kept simple, but also because the data access context – e.g. which application is accessing the data block, which part of the block is actually accessed, and so on – to handle transactional isolation is lost in the block layer. Here, we address the second research question: *what are the implications of pushing transactional isolation to the block layer and what are the required abstractions?*

Due to the portability and compatibility issues, filesystem layers or application layers are not good fits to host transactional APIs. Many systems implemented transactions in such layers [81, 28, 132, 94, 144, 24], and these systems cannot be used universally within cloud storage stacks. Depending on the layer which a new application is developed, these systems can be bypassed or the new application can sit below the systems, which makes the systems unusable. Indeed, no storage system supports ACID transactions from the block layer, which can be potentially used universally. The block storage systems have been used by all applications either directly or indirectly, but have been treated to handle only simple reads and writes. Some systems support transactional atomicity in the block layer [43, 48, 114, 101, 44], but they all come up short supporting full ACID transactions, due to missing transactional isolation support.

A transaction support from the block layer should be very general and easy to use. The API should notify the start and the end of a transaction so that it can be used by any application. To reinforce the missing application context in the block layer, APIs to notify the context should be present. For compatibility among different applications using the same transactional features from the block layer, communication APIs among the applications should be present. All these APIs should work without slowing down applications. We propose and investigate a new design and explore how it can change transactional software development in the cloud storage server. As an instance of this approach, we present Isotope, the first block storage system to support transactional isolation. It works based on a simple API, *beginTX*, *endTX*, and *abortTX*. Data access context is transferred from the application to the system using *mark_accessed* API. The APIs obviate the implementation of transactions, so the application design becomes very simple and easy. By using additional APIs *releaseTX* and *takeoverTX*, different applications can collaborate to work on the same transaction. In Chapter 4, we present the design of Isotope and show how Isotope can facilitate cloud application designs.

1.3.3 Lack of Cloud Storage Server Consistency Control

Regardless of the consistency semantics, a server in a distributed system returns the latest value of a data item in the server. That is, although the value returned by the server may not be the latest globally, the value is locally the latest from the viewpoint of the server. Client-centric consistency allows clients to have independent views of the storage system, such that clients have more choices of servers to access data from and speed up the data access. A client accesses the locally latest value from a server that can be accessed the quickest among the servers that satisfy the client-centric consistency constraints.

The key observation is that the advancement of hardware has made a cloud storage server as powerful and diverse as tens to hundreds of servers in a distributed system from a decade ago. Diverse storage devices lead to various data access speeds within a server, which is similar to accessing different servers with different access latencies in a distributed system. In a sense, the cloud storage server that only returns the latest value stored in one of the storage devices is underutilizing its potentials to provide a finer-grained view of the storage system to the client and to even further speed up the data access. Thus, a cloud storage server is a strong candidate to internally support client-centric consistencies to trade off consistency and performance and to support an even greater variety of fine-grained isolated views of the storage system to the client.

The fundamental challenge is that there has not been any system that supports client-centric consistency within a single server, and applications that can take advantage of client-centric consistency in a server are limited. In addition, to support client-centric consistency, multiple versions of data must exist across different storage devices in the server and new APIs to access versioned data should be present. Even if a server has multiple versions of data, ways to explore and select the data version to return to the client is unknown. Thus, we investigate the research question: *how can client-centric consistency be supported within a cloud storage server to trade off consistency and performance, which systems or applications can adapt to supporting client-centric consistencies, and what are the necessary APIs?*

Client-centric consistency has been supported only in distributed settings [128, 131, 129]. Thus, some applications in distributed domains can work with client-centric consistencies and are likely to take advantage of the client-centric consistency within a single server. A cloud storage server often uses logging for reliability and logging naturally stores multiple versions of data in the server. Storage devices with different access latencies are tiered, e.g. for caching, and sometimes different tiers hold different versions of data as a side effect. Therefore, multiple versions of data are likely to exist in cloud storage servers and there are cases when stale data, which are older versions of data, can be accessed quicker. The similarities between a cloud storage server and a distributed system facilitate the adoption of consistency control and using client-centric consistencies in a single server setting.

To exploring client-centric consistency within a server, we demonstrate the usefulness of trading off consistency and performance. Once the target examples and systems are known we can extract common functionalities and APIs that are necessary. By implementing the functionality and APIs we can truly evaluate whether applying the client-centric consistency within a cloud storage server is feasible and useful. As an instance of this approach, we first identify and study StaleStore, a new class of storage systems which can take advantage of client-centric consistencies within a server and trade off consistency and performance. When different versions of data exist across many different storage devices in a server, StaleStore returns the fastest data given a staleness bound by the client/application. We study the necessary APIs for StaleStores and present a prototype system, Yogurt. Yogurt demonstrates that enabling access to stale data within a server and supporting APIs for a client-centric consistency can lead to better performance. The design principles and the necessary APIs for StaleStore are described in Chapter 5.

1.4 Contributions

By exploring and investigating the three research questions in the previous subsections, we overcome the challenges and contribute towards isolation in a cloud storage system. As problems and questions target various aspects of isolation, the contributions emerge from different angles to collectively achieve an isolated environment in a cloud storage system.

First, **we show how to utilize multiple block devices in cloud storage servers to design a contention-oblivious block storage system based on disks**. We propose a novel chained logging design that logs data over multiple disks in order and a special SSD caching scheme, which protects sequential write operations from reads, to solve the long-standing problems of log-structured designs: this resolves the I/O contention between garbage collection operations and writes as well as the contention between reads and writes. A reduced contention results in better performance isolation, advances the state of the arts, and achieves higher performance in general.

Second, **we explore transactional isolation in the block layer and demonstrate that it can facilitate cloud storage system and application designs.** We show that pushing transactional isolation into the logical block layer can result in simpler high-level storage systems that provide strong transactional isolation semantics without sacrificing performance. Our exploration results in the first system to support transactional isolation from the block layer and the APIs are capable of supporting various applications and programming scenarios. We show that systems and applications can be composed using the API. The API enables applications with different high-level constructs, such as files, directories, and key-value pairs, to work on the same transaction. The system achieves transactional isolation in the cloud.

Third, **we explore how to trade off performance and consistency in a storage server by supporting client-centric consistencies, propose APIs to make use of the trade-off, and define a new class of applications that are capable of utilizing the trade-off.** We first show the feasibility and potential benefit of consistency control within a server by studying existing systems running on storage devices with different speeds. Different from a distributed setting, the consistency control within a server requires careful selection of consistency semantics, as not many applications running on top of a single storage server can tolerate data staleness. We create a class of local storage systems called StaleStores that can support client-centric consistencies by returning stale data for better performance. We describe several examples of StaleStore and show that serving stale data can significantly improve access latencies. Based on this study, we define necessary APIs for providing client-centric consistencies using stale data. We explore the details of how to trade off consistency and performance, and present a prototype system. Using the system, we show that it is possible to provide the client-centric consistency within a server and support users with different views of the storage system to improve the performance.

Overall, all three results of the investigation contribute to transforming a cloud storage system to support isolation. We present how to achieve performance isolation on a disk-based system and how to support transactional isolation from the block layer to facilitate cloud application designs. We show a client-centric consistency can be used to trade off consistency and performance within a server while providing a more flexible view of the storage system. This dissertation leads to a development of cloud storage systems and applications that enable better utilization of a cloud storage server and cloud storage system under isolation.

1.5 Organization

The rest of this dissertation is organized as follows. The scope of the problem and the methodology used for investigating the research questions are described in Chapter 2. Chapter 3 details the exploration for performance isolation in a cloud storage server using contention-oblivious disk arrays. Chapter 4 addresses how we can design transactional APIs inside block storage and facilitate cloud storage systems and applications to achieve transactional isolation. Our study regarding the trade-off between consistency and performance and supporting client-centric consistencies inside a storage server using stale data are presented in Chapter 5. Chapter 6 discusses related work, and Chapter 7 describes future work and concludes.

CHAPTER 2

SCOPE AND METHODOLOGY

This chapter presents the scope of problems which this dissertation addresses and methodologies that are used to investigate the problems. We first characterize the cloud storage server and take a closer look at the problems surrounding isolation. We clarify the scope by reviewing the insights and motivations for each topic and then describe the approach for exploring the problem and the methodology to carry out the investigation.

2.1 Scope: Understanding Cloud Storage Servers

In this section, we describe the scope of the problem. In particular, we enumerate the challenge of each research question in more detail to focus on the approach that we take to investigate the problem. We first explore how disk infrastructures do harm to performance isolation in cloud storage servers and review the opportunities for improvements. Second, we study the need for transactional isolation support from the lower layer of storage stacks. In particular, we revisit the *end-to-end principle* [108], which is a canonical guideline for system designs, to review the soundness of our approach. Finally, we investigate the feasibility of supporting consistency and performance trade-off and examine several existing systems to motivate the need for the support.

2.1.1 Cloud Storage Servers Need Performance Isolation

Cloud storage servers require performance isolation to mitigate unexpected performance fluctuation and degradation. Storage servers have used disks for decades and disks are expected to be around for a long time in the cloud environment [91]. Cloud storage servers host multiple users and the servers inevitably place multiple user workloads on a common disk-based infrastructure. Disks are known to be bad at handling random I/Os, but any workload running on a shared disk can issue random I/Os. Thus, we first conduct a study of how workloads running on shared disks perform and identify the need for performance isolation. Then, we review new storage technologies and potential approaches to achieve performance isolation in cloud storage servers.

Disk Contention

A common example of a cloud storage setting is a virtualized environment, where multiple virtual machines (VMs) execute on a single machine and operate on filesystems that are stored on virtualized disks. The application within each VM is oblivious to the virtual nature of the underlying disk and the existence of other VMs on the same machine. In reality, virtualized disks are implemented as logical block devices or files of the operating system, where the hypervisor runs. While performance isolation across VMs can be achieved by storing each virtual disk on a separate disk, this defeats the goal of virtualization to achieve efficient multiplexing of resources. Accordingly, it is usual for different virtual disks to reside on the same physical disk, and thus, applications accessing the virtual disks concurrently access the same physical disk.

Disk virtualization leads to disk contention. A single application that continually issues random I/Os to a disk can disrupt the throughput of every other application running over that disk [61]. As machines come packed with increasing numbers of cores – and as cloud providers cram more users on a single physical box [142] – it increases the likelihood that some application is issuing random I/Os at any given time, disrupting the overall throughput of the entire system. In fact, throughput in such settings is likely to be sub-optimal even if every application issues sequential I/Os, since the physical disk array sees a mix of multiple sequential streams that is unlikely to stay sequential [65].

To clearly identify these problems, we ran a simple experiment on an 8-core machine with 4 disks configured as a RAID-0 array [97]. RAID-0 stripes data: it splits data into small fixed-size chunks which are written in parallel to the disks. In the experiment, we ran multiple writers concurrently on different cores to observe the resulting impact on throughput. To make sure that the results were not specific to virtual machines, we ran the experiments with different levels of layering: processes writing to a raw logical block device (RAW Disk), processes writing to a filesystem (EXT4 FS), processes within different VMs writing to a raw logical block device (VM + RAW disk), and processes within different VMs writing to a filesystem (VM + EXT4 FS). In the absence of contention (i.e., with a single sequential writer), we were able to obtain 300 to 400MB/s of write throughput in this setup, depending on the degree of layering. Adding more sequential writers lowered throughput; with 8 writers, the system ran at between 120 and 300MB/s.

Figure 2.1 shows the impact on throughput of a single random writer when collocated with sequential writers. We show measurements of system through-

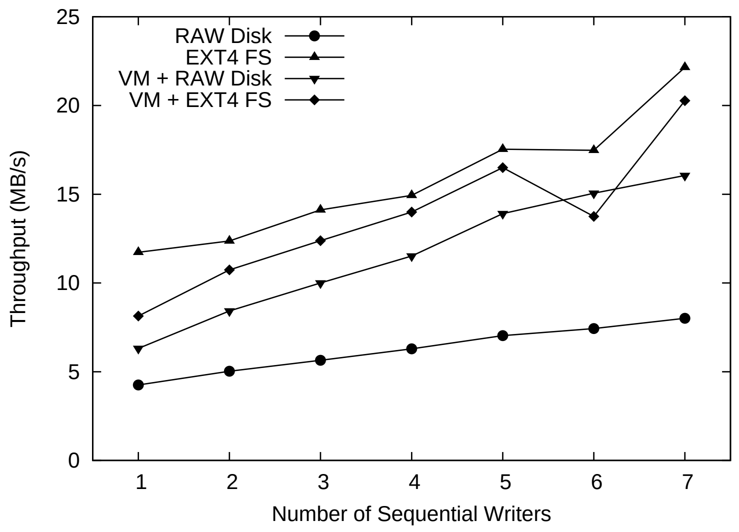


Figure 2.1: Throughput of 4-disk RAID-0 storage under N sequential writers + 1 random writer.

put for increasing numbers of sequential writers, along with a single random writer issuing 4KB writes. For any number of sequential writers and any degree of layering, throughput is limited to less than 25MB/s, representing an order of magnitude drop compared to 300 to 400MB/s throughput without the random writer. The performance drop strongly suggests the need to overcome the limitations of disk characteristics to achieve better performance and performance isolation. At the same time, the observed order of magnitude drop of performance shows that multiple disks under random I/O can perform worse than a single disk under sequential I/O (i.e. 120 MB/s). Thus, our approach to investigate performance isolation is to keep at least one disk always under sequential I/O.

Flash memory and Log-Structured Systems

Log-structured filesystems (LFS) were introduced in the 1990s on the premise that the falling price of random access memory (RAM) would allow for large, inexpensive read caches. Accordingly, workloads were expected to be increasingly write-dominated, prompting designs such as LFS that converted slow random writes into fast sequential writes to disk. A similar approach can be explored using flash memory instead of RAM.

Flash memory price has been steadily dropping [15]. Given this trend, it is tempting to imagine that flash will soon replace disk, or more pragmatically, act as a write cache for disk. The flash write cache can act as an intermediate layer between a RAM and a disk so that users can temporarily write data to the flash quickly and later flush the data to the disk. Unfortunately, cheaper flash translates into less reliable flash, which in turn translates into limited device lifetime [120]. The two ways of lowering flash cost – decreasing process sizes and cramming more bits per flash cell (i.e., multi-level cell (MLC) flash that stores multiple bits per memory cell) – both result in much higher error rates for storing data, straining the ability of hardware error correction code (ECC) to provide disk-like reliability. As a result, lower costs have been accompanied by lower erase cycle thresholds – the number of erase operations permitted for flash memory blocks to store data without errors – and the threshold determines the lifetime of the device when it is subjected to heavy write workloads. In other words, the cost per gigabyte of flash has dropped, but not the cost per erase cycle.

In contrast, read caches, which temporarily store data from disks and send the stored data quickly to the reader, are a more promising use of flash. Unlike primary stores or write caches, read caches do not need to see every data update immediately, but instead have leeway in deciding when (and whether) to cache data. For example, a read cache might wait for some time period before caching a newly written block so that writes on the same block can be coalesced in the meantime and flash lifetime can be extended by skipping the writes. It could also avoid caching data that is frequently overwritten but rarely read. Crucially, read caches do not need to be durable and hence the lower reliability of flash over time is not as much of a barrier to deployment; the read cache only requires a reliable mechanism to detect data corruption, which effectively translates into a cache miss.

Accordingly, our core assumption is nearly identical to that of the original LFS work: larger, effective (flash-based) read caches will result in write-dominated workloads. Unfortunately, simply using LFS under a flash-based read cache does not work, because of two key problems. First, as noted earlier, LFS is notorious for its garbage collection (GC); GC reads (which are unlikely to be caught by a read cache) can contend with writes from applications, negating the positive effect of logging writes. Second, even a small fraction of random reads past the cache can interfere with write throughput. In other words, LFS effectively prevents write-write contention but is very susceptible to read-write contention, both from GC reads and application reads. Our goal is to build a log-structured storage design that prevents both write-write as well as read-write contention.

In addition to caching reads, flash memory further acts as a catalyst for log-structured designs by providing an inexpensive, durable metadata store. Metadata refers to information about data including size, location, accessed time, and so on. A primary challenge for any log-structured system involves maintaining an index over the log, which maps the location of (the latest) data blocks. As a result, log-structured designs are usually found at layers of the stack that already require indices in some form, such as filesystems or databases. Designs at the block-level with a logging component have historically suffered from seeks on on-disk metadata, or predicated on the availability of battery-backed RAM or non-volatile-RAM (NV-RAM) [43, 48, 141]. Consequently, such designs have been restricted to expensive enterprise storage solutions. By providing an inexpensive means of durably storing an index and accessing it rapidly, flash enables log-structured designs at lower layers of the stack, such as the logical block device.

Following the trend of increased use of flash memories [91], we explore performance isolation on disks using flash and logging in Chapter 3.

2.1.2 Cloud Storage Servers Need Transactional Isolation

Transactional isolation support from the cloud storage server would benefit most applications and systems running on top of the server, because the cloud inherently entails concurrent data accesses. Cloud storage servers host a variety of applications and systems, but most applications and systems implement transactional isolation of their own. Thus, transactional isolation has become a redundant feature that is used by many but not supported by the cloud storage server itself. Transactional isolation has become redundant because the storage stack is traditionally designed to place sophisticated functionalities, including transactions, in the high layers of the stack [79, 108]. However, the cloud opens up new opportunities to question the traditional storage stack design. The logical block device layer is the lowest common software layer of the storage stack which has been kept simple but it is used directly or indirectly by most systems and applications. To support transactional isolation (in addition to atomicity and durability) as a feature provided from the cloud storage server to all systems and applications, we reason about supporting transactions from the block layer and explore the potential benefit.

End-to-End Argument

End-to-end argument [108] is a system design guideline which helps the designer to decide where to place a particular functionality in a layered system. The argument advocates placing functionalities in the end-application or the high layers of the software stack in two cases. The first is when application-specific care or information is necessary even after the low layer has processed the functionality. Exceptions can be made when there is a performance or utility reason to place the functionality down the stack. The second is when placing the functionality in the low layer incurs unnecessary overhead to applications that do not use the functionality. To summarize, if a functionality is not complete by itself in the low layer, is not usable for most applications in the system, or does not have performance benefits to be placed in the low layer, it should be located in the high layers of the stack. We carefully review the end-to-end argument to investigate the soundness of transactional isolation support from the block layer.

Although the first part of the end-to-end argument may not completely comply with our approach, the exception to the first clause ensures that transactional isolation in the block layer is a viable approach especially in cloud storage servers. Transactional isolation has been implemented in high layers of the storage stack, partially because of the first part of the end-to-end argument. Transactional isolation requires handling of information about which part of data the application has accessed in a transaction. However, transactional isolation in the block layer of cloud storage servers passes the first clause of the end-to-end argument, because the block layer support is undeniably useful for most applications running on a cloud storage server. The applications require transactional isolation or concurrency control by default in the cloud and once the necessary information for a transaction becomes available to the block layer, applications do not have to handle transactions redundantly. One of our goals is to provide a general block level API for transactions, so applications can easily adopt transactional features from the block layer.

Considering the second part of the end-to-end argument, transactional isolation is efficiently implementable at a low layer of the stack with negligible performance overhead using flash based storage devices, terabytes of RAM, and tens to hundreds of cores that already exist in cloud storage servers. Moreover, the fact that most applications require transactional isolation in the cloud eliminates concerns for imposing unnecessary performance overhead to any application.

Other Needs and Benefits

In addition to the examination of the end-to-end argument, we investigate advantages and other goals for support transactional isolation from the block layer:

Overcoming the complexity of locks: Storage systems typically implement pessimistic concurrency control via locks, opening the door to a wide range of aberrant behavior such as deadlocks. A deadlock is a status which a program cannot make any progress because processes/threads in the program have locked different data simultaneously and are indefinitely waiting for others to release the lock. This problem is exacerbated when developers attempt to extract more parallelism via fine-grained locks, and add more complexity by incorporating mechanisms for atomicity and durability [87]. Transactions can provide a simpler design of storage system by supplying isolation, atomicity and durability at the same time using a single abstraction.

Supporting a generic transaction: Storage systems often provide concurrency control APIs over their high-level storage abstractions; for example, NTFS, a Windows filesystem, offers transactions over files, while Linux provides file-level locking. Unfortunately, these high-level concurrency control primitives often have complex, weakened, and idiosyncratic semantics [98]; for instance, NTFS provides transactional isolation for accesses to the same file, but not for directory modifications, while a Linux lock using *fnctl* commands can be released when the file is closed by any process that was accessing the file instead of an explicit unlock [5]. The complex semantics are typically a reflection of a complex implementation, which has to operate over high-level constructs such as files and directories. In addition, if each storage system implements isolation independently transactions cannot span over different systems: for example, it is impossible to do a transaction over a file on NTFS and an arbitrary database system. One of our goals for exploring block level transactions is to support transactions over multiple systems that work on different data constructs.

Efficient transactions using multiversion concurrency control. Pessimistic concurrency control with locks is slow and prone to bugs; for example, when locks are exposed to end applications directly or via a transactional interface, the application could hang while holding a lock. Optimistic concurrency control [74] works well in this case, ensuring that other transactions can proceed without waiting for the hung process. Multiversion concurrency control works even better. Multiversion concurrency control (MVCC) is one of the optimistic concurrency control mechanisms which maintains multiple versions of data to serve users with different snapshots. Transactions with stable, consistent snapshots (a key property for arbitrary applications that can crash if exposed to inconsistent snapshots [59]) allow read-only transactions to always commit [38] and enables weaker but more performant isolation levels such as snapshot isolation [36].

However, implementing MVCC can be difficult for storage systems due to its inherent need for multiversion states. High-level storage systems are not always intrinsically multiversioned, making it difficult for developers to switch from pessimistic locking to a MVCC scheme. Multiversioning can be particularly difficult to implement for complex data structures like B-trees – a balanced tree commonly used in databases and filesystem to index data blocks – requiring explicit marking of deleted data which is known as tombstone [53, 103].

In contrast, multiversioning is relatively easy to implement over the static address space provided by a block store (for example, no tombstones are required since addresses can never be deleted). Additionally, many block stores are already multiversioned in order to obtain write sequentiality: examples are shingled drives [26], SSDs, and log-structured disk stores. Thus, as an efficient implementation strategy for transactions, we investigate pushing MVCC in the block layer as well.

Chapter 4 details how we design transactional block layer using a new multi-version concurrency control method and demonstrate how this facilitates cloud application designs.

2.1.3 Cloud Storage Servers Need Consistency Control

Distributed systems use client centric consistency semantics to trade off consistency and performance and a cloud storage server, which has become as powerful as a distributed system, has the potentials to take advantage of the same trade-off. Cloud storage servers are highly parallel internally and use storage devices with different speeds. Tens to hundreds of CPU cores and tens of heterogeneous storage devices are similar to servers in distributed systems, and distinct access speeds of storage media are analogous to network delays between the servers. However, no study has been conducted surrounding the trade-off between consistency and performance within a cloud storage server. Supporting client-centric consistency within a server can potentially provide each user with an independent view of the storage server under better performance. By exploring the similarity between a cloud storage server and a distributed system, we investigate how to trade-off consistency and performance in the cloud storage server.

Besides the similarity of hardware between a cloud storage server and a distributed system, our study for consistency and performance trade-off within the cloud storage server relies on two key observations: local storage systems often have multiple versions of data due to logging and caching, and older versions, which are known as stale data, are often faster to access. We first enumerate the example systems where the observations hold and investigate why older versions are faster to access.

Because there are no systems that already trade-off consistency and performance within a server, we built high-fidelity emulations of three systems to further study the characteristics of the example systems and evaluate the potentials for the consistency-performance trade-off. The emulations are not functionally complete (e.g., they do not handle recovery after crash) but faithfully mimic the I/O behavior of the original system. Using these emulated studies, we show that accessing older versions can significantly cut access latency while achieving finer-grained control of client-centric consistencies in a server.

S1. Single-disk log-structured stores. The simplest and most common example of a system design that internally stores faster stale versions of data is a log-structured storage system, either in the form of a filesystem [107] or block store [48]. Such systems extract sequential write bandwidth from hard disks by logging all updates. This log-structured design results in the existence of stale versions; furthermore, these stale versions can be faster to access if they are closer to the disk arm than the latest version. Previous work has explored storing a single version of data redundantly and accessing the closest copy [147].

S2. SSD FTLs (Flash Translation Layers). SSDs based on NAND flash are internally log-structured and the FTL, a software in flash chips or SSDs, is in charge of maintaining the log, redirecting I/O requests according to the log index, and performing garbage collections [120]. Data in flash are written in an *erase block*, which consists of multiple 4KB pages, and the pages in the same erase block are erased or reset together during garbage collection. Stale versions, which are created by logging, can be faster to access if the latest version happens to be in an erase block that is undergoing garbage collection.

S3. Log-structured arrays. Some designs chain a log over multiple disks. Gecko which we explore in Chapter 3 is a storage array with a chained log; updates proceed to the tail drive of the log, while reads are served by all the disks in the log. In such a design, reads from disks in the body of the log are faster since they do not interfere with writes. Accordingly, reading a stale version in the body of the chained log may be faster – and less disruptive to write throughput – than reading the latest version from the tail drive.

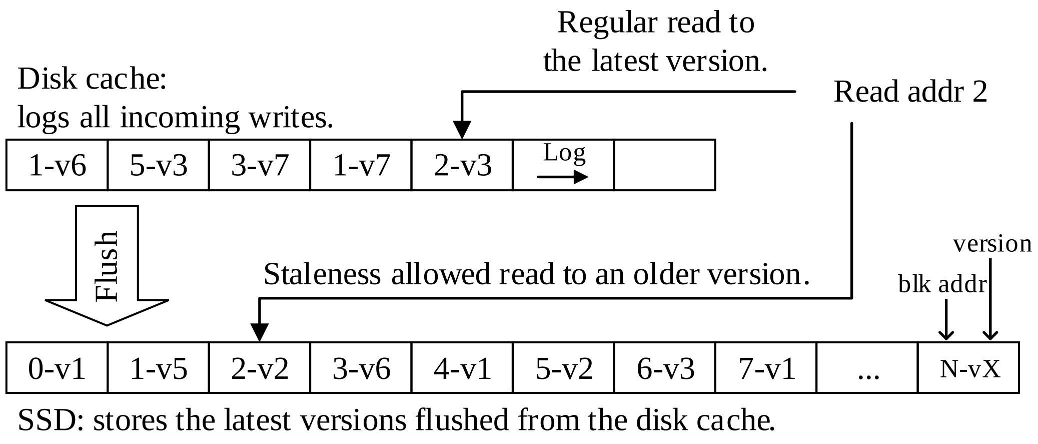
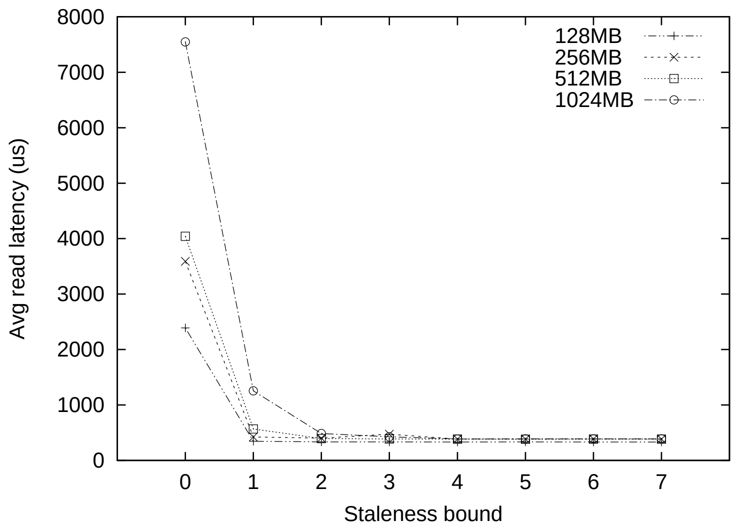
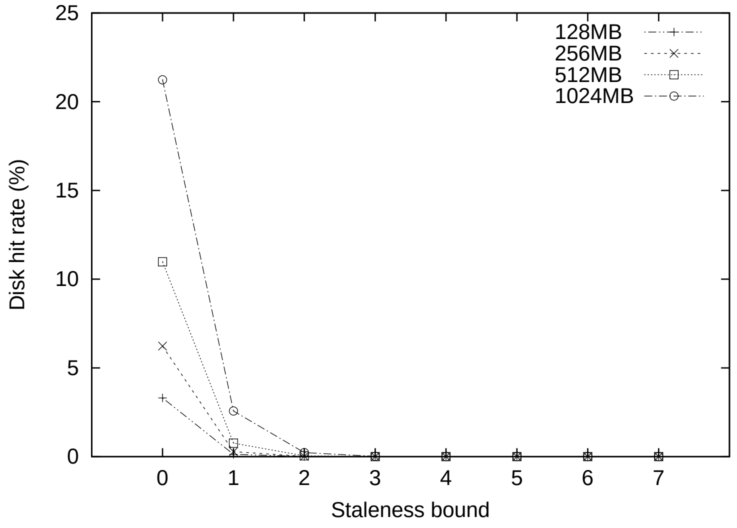

Regular read to the latest version.
Disk cache: logs all incoming writes.
Read addr 2
1-v6
5-v3
3-v7
1-v7
2-v3
Log
Flush
version
Staleness allowed read to an older version.
blk addr
0-v1
1-v5
2-v2
3-v6
4-v1
5-v2
6-v3
7-v1
...
N-vX
SSD: stores the latest versions flushed from the disk cache.

Figure 2.2: In the Griffin system, being able to read older versions from a SSD than the latest version from the disk cache can be faster.

S4. Durable write caches that are fast for writes but slow for reads. Griffin [124] layers a disk-based write cache over an SSD; the goal is to coalesce overwrites before they hit the SSD, reducing the rate at which the SSD wears out. In such a system, the latest version resides on the write cache; reading it can trigger a slow, random read on the disk that disrupts write throughput. On the other hand, older versions live in the backing SSD and are much faster to access (Figure 2.2).



(a) Read latency



(b) Disk cache hit rate

Figure 2.3: Read latency and disk cache hit rate of Griffin with different disk to SSD data migration trigger sizes. Accessing stale data can avoid reading from the disk.

We implemented an emulator for the Griffin system. Figure 2.3-(a) shows the latency benefit of serving older versions. The y-axis is the latency; the x-axis is the parameter for the bounded staleness consistency guarantee, signifying how stale the returned value can be in terms of the number of updates it omits. We run a simple block storage workload where a 4GB address space is written to and 8 threads issue random reads and writes with 9 to 1 ratio. Depending on the configuration, the Griffin system flushes data from the disk cache to SSD whenever 128MB to 1GB worth of data is written to the disk. The figure shows that allowing the returned value to be stale by even one update can reduce read latency down to 1/8 and down to 1/20 by allowing values stale by four updates. Figure 2.3-(b) shows the ratio of reads hitting the disk cache. Read accesses to disk can be eliminated by allowing values stale by five updates.

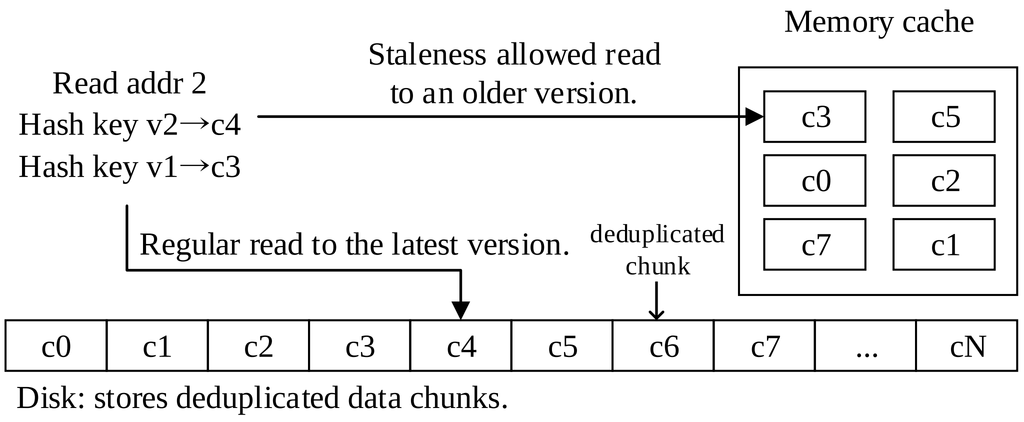
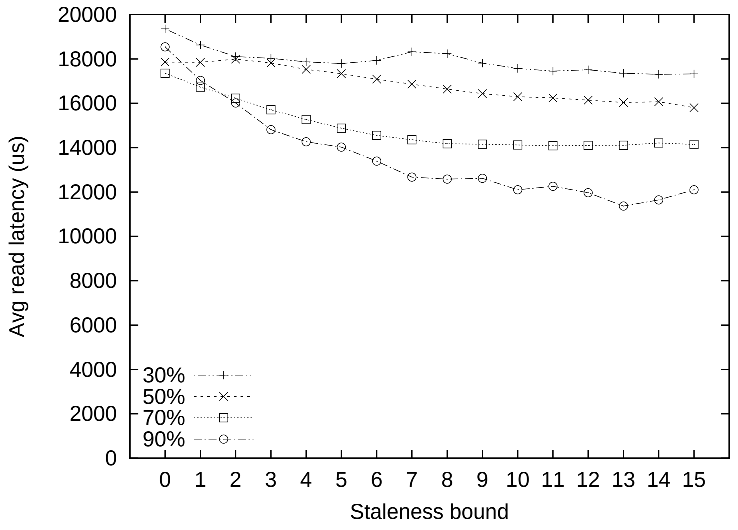

Memory cache
Staleness allowed read to an older version.
Read addr 2
Hash key v2→c4
Hash key v1→c3
c3
c5
c0
c2
c7
c1
Regular read to the latest version.
deduplicated chunk
c0
c1
c2
c3
c4
c5
c6
c7
...
cN
Disk: stores deduplicated data chunks.

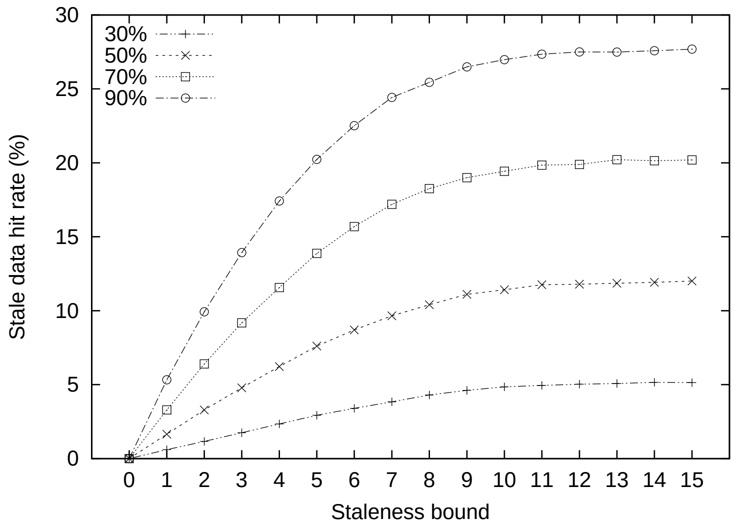
Figure 2.4: In a deduplicated system with cache, data items are shared with many others. If an older version is referenced by another address and is inside the cache, reading this than the latest version is in the disk is faster.

S5. Deduplicated systems with read caches. Deduplicaiton finds redundant copies of information in a storage system to saves space: redundant copies are removed and only one copy is shared by all users of the system [86]. In such systems, an older version of a data item may be identical to the latest, cached version of some other data item; in this case, fetching the older version can result in a cache hit (Figure 2.4).

Figure 2.5 shows the performance and memory cache hit rate on stale data of



(a) Read latency



(b) Cache hit rate on stale data

Figure 2.5: Read latency and memory cache hit rate on stale data in a deduplicated system with different deduplication rates. Accessing stale data results in higher cache utilization and lower read latency.

a deduplication system which is under bounded staleness guarantees. The system is a block store which deduplicates 4KB blocks. 8 threads randomly read and write blocks with 9 to 1 ratio within 4GB address space. The overall deduplication ratio, which defines the amount of redundant data, is controlled to be

30 to 90 percent. The system uses a disk as a primary storage and 256MB RAM as a read/write cache, which is indexed by the hash key and uses least recently used (LRU) policy. LRU policy evicts least recently accessed data first when the cache is full. The performance improvement plateaus as the allowed staleness bound increases, but the performance is improved up to 10 to 35% depending on the deduplication ratio (Figure 2.5-(a)). Such performance improvement trends follow the cache hit rate on stale data (Figure 2.5-(b)).

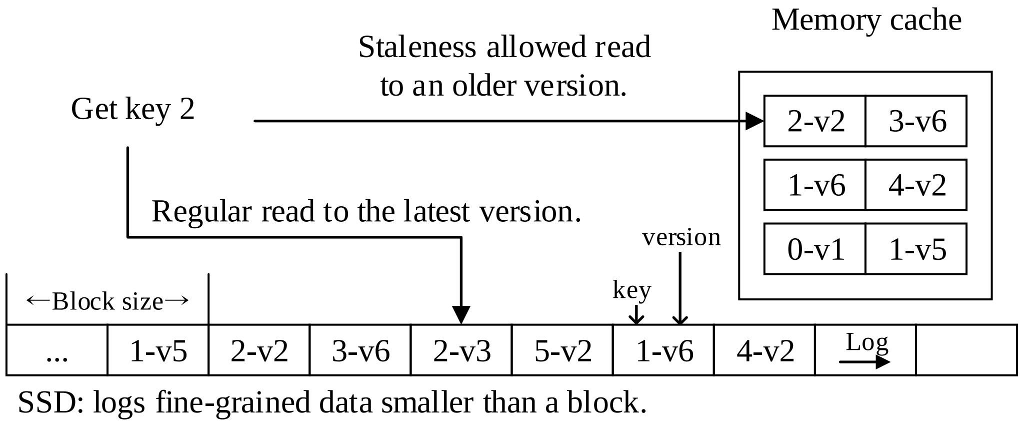
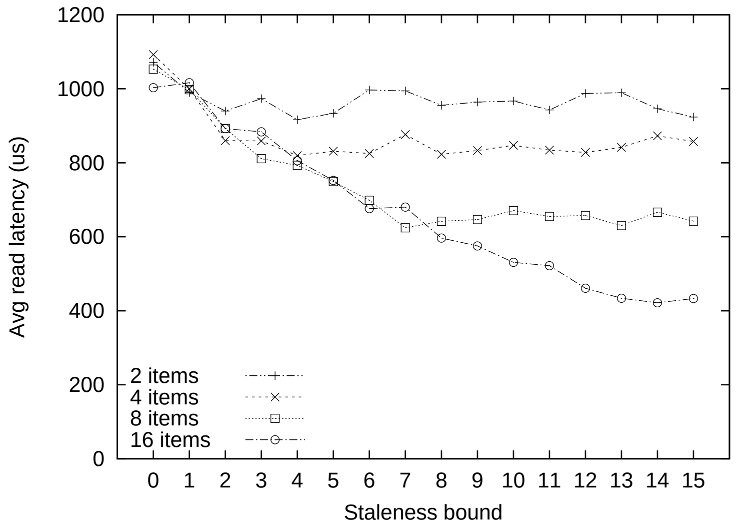
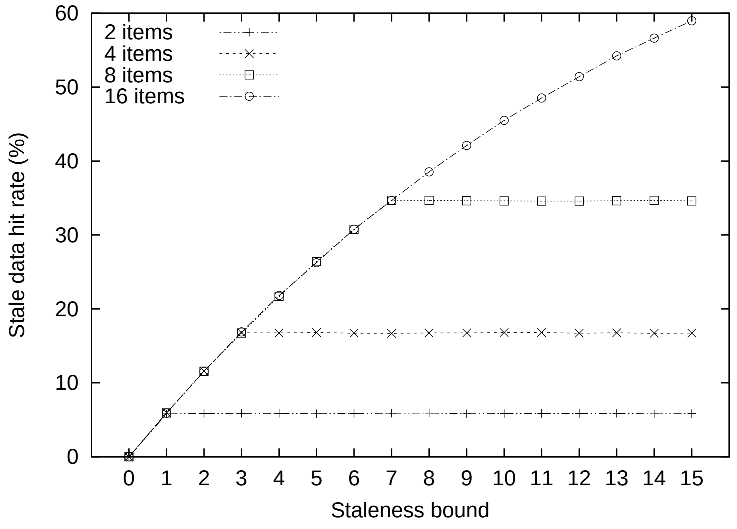

Memory cache
Staleness allowed read to an older version.
Get key 2
2-v2
3-v6
1-v6
4-v2
0-v1
1-v5
Regular read to the latest version.
version
key
←Block size→
...
1-v5
2-v2
3-v6
2-v3
5-v2
1-v6
4-v2
Log
SSD: logs fine-grained data smaller than a block.

Figure 2.6: If data items are smaller than the cache block in a fine-grained logging system, other items (e.g. 2-v2) can follow an item (e.g. 3-v6) being read into the cache. If the item that followed is an older version, accessing the older item can be faster than the latest item in the SSD.

S6. Fine-grained logging over a block-grain cache. Consider a log-structured key-value store implemented over an SSD (e.g., like FAWN [30]), which in turn has an internal memory cache. New key-value updates are merged and written as single blocks at the tail of a log layered over the SSD's address space. As key-value pairs are overwritten, blocks in the body of the log hold progressively fewer valid key-value pairs, reducing the effectiveness of the memory cache within the SSD. However, if stale values can be tolerated, the effectiveness of the memory cache increases since it holds valid versions – stale or otherwise – for a larger set of keys (Figure 2.6).



(a) Read latency



(b) Cache hit rate on stale data

Figure 2.7: Read latency and memory cache hit rate on stale data in system with fine-grained logging over a block-grain cache. The read latency decreases and the cache hit rate increases when the allowed staleness of data and the number of items placed in a cache block increase.

Figure 2.7 shows the performance of a simple emulation of a key-value store layered over an SSD with a 256MB DRAM cache. 1 million key-value pairs are present in the system and 8 threads randomly read and write them with 9 to 1 ratio. The key-value pair size is parameterized such that 2 to 16 pairs can be stored in a block. If *N* key-value pairs fit in a block, at most *N* − 1 key-value pairs can be wastefully loaded in the cache. Allowing access to older versions reenables utilization of potentially wasteful data items in the cache block (Figure 2.7-(b)) and higher utilization of DRAM cache results in increased performance up to 60% (Figure 2.7-(a)).

S7. Systems storing differences from previous versions. Some log-structured systems store new data as changes against older versions. In Delta-FTL [145], when a block is overwritten, the system does not log the entire new block; instead, it logs the change against the existing version. For instance, if only the first 100 bytes of the block have been changed by the overwrite, the system only logs the new 100 bytes. In such a system, accessing the latest version requires reading both some old version and one or more changes, triggering multiple block reads. Accessing an older version can be faster since it requires fewer reads.

All the above cases keep older versions to achieve better performance, storage durability, storage utilization, ease of data maintenance, and so on, but being able to access older versions faster than the latest version is a side effect. However, our exploration has shown that trading off consistency and performance within a server has potentials to improve system performance. In Chapter 5 we investigate a system that explicitly utilizes this side effect – whenever it is possible – to speed up storage performance and to provide fine-grained control of client-centric consistency inside a server.

2.2 Methodology

In this dissertation, we investigate how to achieve isolation in cloud storage servers from different angles. Each angle stems from different research questions and challenges surrounding performance, transaction, and consistency control. We explore each topic by designing, implementing, and evaluating a system. The system design and implementation represent our approach to investigate the research question, and the evaluation reveals our findings and verifies the soundness of our approach.

In the rest of the section, we describe the environment, where we build our systems, and detail the tools and applications that we use to evaluate the systems. We also present baseline systems that we employ to compare our approaches.

2.2.1 Linux Device Mapper

We built three systems to investigate isolation in cloud storage servers. All our systems are built in a Linux kernel as device mappers. The device mapper is a Linux kernel module that enables logical block device management [12]. The device mapper is located above or in the same storage stack as the logical block device as shown in Figure 2.8: it is under the filesystem and above the device driver. Similar to how block devices are represented and accessed in a Linux kernel, the device mapper shares the same block I/O interface. Filesystems can run on top and interact with the device mapper using block I/Os and applications can also directly open and access the device mapper. Examples of a device

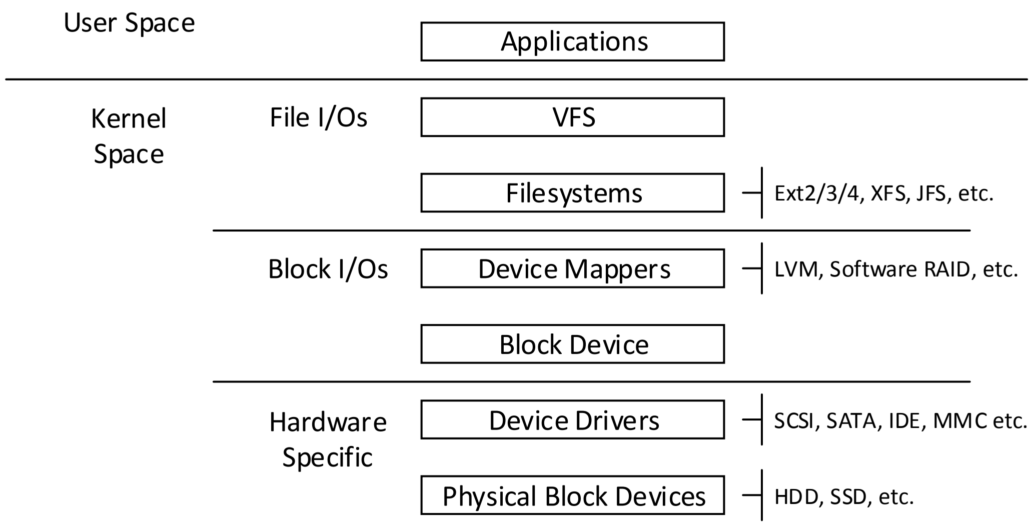


Figure 2.8: Linux storage stack and device mapper.

mapper are logical volume manager (LVM) and software RAID. LVM creates resizable logical disks that are smaller than or larger than an actual disk using a part of a disk or multiple disks and software RAID creates a logical disk that uses RAID techniques using multiple disks under the hood.

The device mapper module can combine multiple physical or logical block devices to create a new logical block device. Once a device mapper creates a logical block device, it maps the I/O request coming into the block device to the underlying block device. For example, LVM can create a large size logical block device by combining multiple physical block devices and redirects the I/Os to the physical block device. Similarly, the systems in this dissertation are represented as a logical block device, but internally log data to multiple disks and cache data in a SSD or memory. In addition to simply creating different sized block devices and redirecting the I/Os, device mappers are used to implement more sophisticated functionalities, such as encryption (`dm-crypt`), deduplication (`dm-dedup`), and so on.

Because a device mapper represents logically the same form as regular logical block devices, it can even be nested under each other. For example, a logical volume created by the LVM can be used under software RAID. In Chapter 3, we use this feature for evaluation. We use multiple disks to create a RAID 0 volume and run our system and baseline systems on top.

The `ioctl` system call is used to trigger an interactive function that is specific to a device mapper. The device mapper's programming interface includes handling of custom `ioctl` calls. We use the `ioctl` interfaces to implement new APIs and use transactional and consistency trading off functionalities.

Device mappers are configured and controlled using `dmsetup` and `libdevmapper`, which are a user space tool and a library, respectively. For example, `dmsetup`'s `message` command can reconfigure the device mapper or issue specialized commands on the fly, and its `status` command can retrieve the status information of the device mapper.

2.2.2 Emulations

To investigate our approaches that do not strictly require a full-fledged system, we use emulations. Emulations are usually run in user space and are easier to implement than an actual system in an operating system (OS) kernel. Emulations enable fast prototyping and can yield a very precise outcome that is almost identical to running a real system. An emulated system may or may not be able to run applications that used to run on the original system, but given the input, the core outcome of interest should be identical. In this dissertation, we use emulations focusing on the I/O behaviors of the system. The emulations do not fully emulate the original system's behavior and run real applications, but the emulations trigger the same read/write operations to the block devices as the real system when I/O workloads are executed on top.

Emulations are suitable to test the performance and to analyze the behavior of storage systems. Accessing a persistent storage media takes longer than accessing any other hardware parts in a computer system. While access to main memory takes hundreds of nanoseconds, accesses to a SSD and a disk take hundreds of microseconds and tens of milliseconds, respectively, which results in three to five orders of magnitude gaps. Thus, computation time in CPUs and memory access time are easily hidden by the storage access time. Namely, differences in CPU and memory accesses between a real system and an emulated system are not easily noticeable as long as the I/O access patterns are identical. For this reason, emulating a kernel storage system in a user space does not cause a huge difference regarding performance, and doing so makes the testing and prototyping process easier and faster as debugging in kernel space can be very time-consuming. We implemented all of our systems in the Linux kernel, but used emulations to quickly explore and test new functionalities that can be time-consuming to implement in the kernel. We use emulations in Chapter 3 to evaluate some additional functionalities that do not exist in kernel implementations.

Emulations can be useful when the internals or source code of a system are not accessible. One can quickly mimic and emulate the known core behavior of the system, which others have built, to get internal statistics or to compare against other systems. For example, it is difficult to openly access the firmware code or the hardware implementation of SSDs. In Chapter 3, we emulate the behavior of an SSD to test the lifetime expectancy when it is used as a cache. In Section 2.1.3, we have already shown several emulations of systems designed by others to prove our motivating concepts that returning stale data can result in faster data accesses and providing consistency and performance trade-off can be useful.

2.2.3 Public Workloads and Systems

To evaluate our systems and approaches, we use block I/O traces, benchmarks, storage systems, and development frameworks. In this subsection, we describe the ones that are publicly available and how we use them.

Block I/O Traces

Block I/O traces are often used to evaluate storage systems. Because the traces only describe the I/O access patterns and not the data contents that can be sensitive, block I/O traces are relatively easier to release in the public domain. Microsoft has released three sets of traces – Microsoft production server traces, Microsoft enterprise traces, and Microsoft Research Cambridge traces – in the SNIA repository [20] and they have been actively used in many research projects and papers. The traces are collected from SQL servers, Exchange mail servers, network filesystems, Live map service, printer servers, source management servers and so on, which ranges from a small system to a large distributed scale system. These are very useful as they are collected from a real deployment of systems and were collected for an extended time period. These traces can be used to evaluate how a new system can perform under various workload patterns that can be found in real life. By running different I/O traces simultaneously, we emulate a cloud environment where random traces run together. In this dissertation, the block I/O traces are mostly used for evaluating performance isolation.

Benchmarks

Using a benchmark software is a standard way of evaluating a new system. Storage system benchmarks generate various I/O patterns and enable testing the system under different settings.

IOZone [9]: IOZone is a filesystem benchmark tool. It generates various I/O patterns in different phases such as sequential reads, sequential writes, re-writes, read backwards, read strided, random reads, random writes, asynchronous reads, and asynchronous writes. Although it is a filesystem benchmark, block devices can be tested when IOZone is run directly on the logical block layer. For the evaluation of our device mapper based systems, we used options using `O_DIRECT` and `O_SYNC` which enable I/Os to directly interact with the block layer. At the end of the execution, IOZone returns statistics including bandwidth and latency. Using multiple IOZone threads, we create a system environment with multiple concurrent users issuing heavy I/O.

LevelDB benchmark [11]: LevelDB is a library key-value store developed by Google. LevelDB benchmark is designed as a part of LevelDB to evaluate the performance. LevelDB benchmark uses the LevelDB interface to issue I/O requests. Similar to IOZone, the benchmark generates synthetic I/O patterns, such as random and sequential reads and writes. We use this benchmark and

LevelDB to evaluate our approach in Chapter 4. We build our own key-value store, which shares the LevelDB interface, on top of our system and compare the performance against LevelDB.

Yahoo! Cloud Serving Benchmark (YCSB) [46]: YCSB is developed by Yahoo to evaluate key-value stores and cloud serving stores. It mostly generates I/Os based on zipf distributions [41], where I/Os appear more on popular data items and the popularity is logarithmically distributed to each data item. There are six basic workload patterns: update heavy, read mostly, read only, read latest, short ranges, and read-modify-write. We use update heavy workload (YCSB workload-a) which issues 50% read and 50% write I/Os in zipf distributions. Similar to the LevelDB benchmark, we use it on top of key-value stores running on top of our systems in Chapter 4 and 5 to evaluate the performance.

FUSE Framework

Filesystem in user space (FUSE) framework [6] lets developers write user space filesystems. It consists of a FUSE kernel module and the `libfuse` user space library that connects the user space file system implementation with the kernel module. Typically, a filesystem in the kernel space is difficult to develop and test, due to the inherent difficulty of kernel programming, but FUSE reduces this burden. Although there are overheads for running a filesystem in the user space due to frequent context switching between kernel space and user space, FUSE is used for many purposes, such as for prototyping and education. In Chapter 4, we build a transactional filesystem using FUSE to evaluate our system.

Baseline Systems

To support our claim and compare our approaches, we use a baseline configuration or a baseline system created by others. The following systems and configurations are the publicly available baselines we used for the investigation.

Software RAID [13]: Linux software RAID is a device mapper module that configures multiple block devices into a RAID drive. It supports RAID 0, 1, 4, 5 and 6 [97]. We use RAID-0 configuration to parallelize multiple block devices and to run logging on top. We use this configuration to evaluate our system in Chapter 3.

LevelDB [10]: LevelDB is a library key-value store optimized for range queries. It uses a log-structured merge (LSM) tree [95] to maintain its data. The LSM tree has several levels of logs and sorts the index in each level while key-value pairs are being inserted. LSM tree makes LevelDB perform fast writes and execute range queries efficiently. We use LevelDB and LevelDB benchmark to compare and evaluate the performance of our system in Chapter 4.

Ext2/Ext3 Filesystem [64]: Ext2 and ext3 filesystems are widely used filesystems in Linux. Ext2 is simple and fast but lacks fault tolerance, so ext3 was developed to overcome this problem by logging the changes to be made (i.e. journaling). We run ext2 and ext3 filesystems on top of our systems to evaluate the performance and overhead.

2.3 Summary

A cloud storage server faces challenges to support performance isolation, transactional isolation, and client-centric consistency control. First, the cloud storage needs to overcome the characteristics of spinning disks, which are susceptible to concurrent accesses, to accomplish performance isolation. Opportunities lie in logging, new storage technologies such as SSDs and tens of storage devices in cloud storage servers. Second, the storage stack has not supported rich functionalities from the low layers and the cloud environment opens up new opportunities to redesign the stack. Support of the transactional from the block I/O layer suggests strong potentials to preclude the redevelopment of transactions in the cloud and to grant compatibility of transactions. Finally, the cloud storage lacks support for control of consistency inside servers for consistency and performance trade-offs. Carefully exploring the distinct data access latencies in servers and providing APIs for client-centric consistency have the potentials to further boost the performance and support better-isolated views of the storage system to the users. In the remainder of this dissertation, we describe how we investigate the research questions that we asked and present the system designs, implementations, and evaluations that realize our approach and validate our findings.

CHAPTER 3

PERFORMANCE ISOLATION WITH CONTENTION-OBLIVIOUS DISK ARRAYS

To provide isolation of performance in cloud storage servers, the concurrent nature of users must be understood and the underlying storage device characteristics must be overcome. Cloud environments often use virtualization, where the compute and storage resources of each physical server are multiplexed across a large number of applications. The cloud entails multiple users who are routinely assigned to different computational processing (CPU) cores on the same server. The increasing number of cores on individual servers forces applications to co-exist on the same machine. The hardware cost to host multiple users and applications drops by sharing of resources, but resource sharing leads to greater challenges for performance isolation, especially on disks.

When many applications run together, applications are susceptible to the behavior of other applications executing on the same machine. In particular, contention in the storage subsystem of a cloud storage server is a significant issue, especially when a disk array is shared by multiple applications running on different cores. In such a setting, an application designed for high I/O performance – for example, one that always writes or reads sequentially to disk – can perform poorly due to random I/O introduced by applications running on other cores [61]; in Chapter 2, we quantified this effect and showed how disks are poor at performance isolation. In fact, even in the case where every application on the physical machine accesses storage strictly sequentially, the disk array can still see a non-sequential I/O workload due to the intermixing of multiple sequential streams [65]. Disk contention of this nature is endemic to any system design where a single disk array is shared by multiple applications running on different cores. Thus, better performance isolation is necessary for disk-based cloud storage servers.

Existing solutions to mitigate the effects of disk contention revolve around careful scheduling decisions, either spatial or temporal. For instance, one solution to minimize interference involves careful placement of applications on machines [61, 62]. However, this requires the cloud provider to accurately predict the future I/O patterns of applications. Additionally, placement decisions are usually driven by a large number of considerations, not just disk I/O patterns; these include data/network locality, bandwidth and CPU usage, migration costs, security concerns, etc. A different solution involves scheduling I/O to maintain the sequentiality of the workload seen by the disk array. Typically, this involves delaying the I/O of other applications while a particular application is accessing the disk array. However, I/O scheduling sacrifices access latency for better throughput, which may not be an acceptable trade-off for many applications.

A more promising approach is to build systems that are oblivious to contention by design. For instance, log-structured designs for storage – such as the log-structured filesystem (LFS) [107] – can support sequential or random write streams from multiple applications at the full sequential speed of the underlying media. Unfortunately, the Achilles' Heel of LFS is read-write contention caused by garbage collection (GC) [116, 82]; specifically, the random reads introduced by GC often interfere with first-class writes by the application, negating any improvement in write throughput. Additionally, LFS can also be subject to read-write contention from application reads; the original LFS work assumed

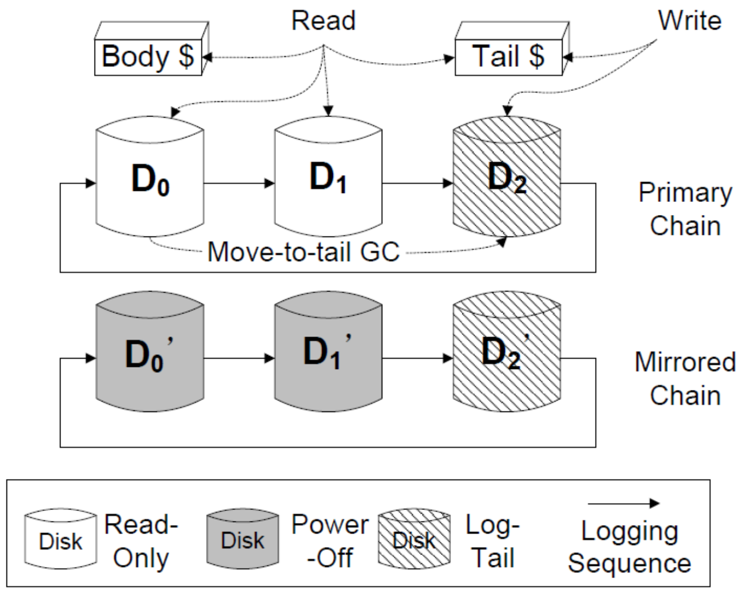


Figure 3.1: Chained logging: all writes go to the tail drive of the chain, while reads are serviced mostly from the body of the chain or a cache. Mirrors in the body can be powered down.

that large caches would eliminate reads to the point where they did not interfere with write throughput. More recently, systems have emerged that utilize new flash technology to implement read caches or log-structured write caches [42] that can support contention-free I/O from multiple applications. However, this results in a highly stressful write workload for the flash drives that can wear them out within months [124].

In this chapter, we present Gecko, a new log-structured design for disk arrays. The key idea in Gecko is *chained logging*, in which the tail of the log – where writes occur – is separated from its body by placing it on a different drive. In other words, the log is formed by concatenating or chaining multiple drives. Figure 3.1 shows a chain of three drives, D_0, D_1 and D_2. On a brand new deployment, writes will first go to D_0; once D_0 fills up, the log spills over to D_1, and then in turn to D_2. In this state, new writes go to D_2, where the tail of the log is now located, while reads go to all drives. As space on D_0 and D_1 is freed due to overwrites on the logical address space, compaction and garbage collection is initiated. As a result, when D_2 finally fills up, the log can switch back to using free space on D_0 and D_1. Any number of drives can be chained in this fashion. Also, each link in the chain can be a mirrored pair of drives (e.g., D_0 and $D_0^{'}$) for fault-tolerance and better read performance.

The key insight in chained logging is that the sequential, contention-free write bandwidth of a single drive is preferable to the randomized, contention-affected bandwidth of a larger array. As with any logging design, chained logging ensures that write-write contention between applications does not result in degraded throughput, since all writes are logged sequentially at the tail drive of the chain. Crucially, chained logging also eliminates read-write contention between garbage collection (GC) activity and first-class writes by separating the tail of the log from its body. In the process, it trades off the maximum contention-free write throughput of the array – which is now limited to the sequential bandwidth of the tail drive of the chain – in exchange for stable, predictable write performance in the face of contention. In our evaluation, we show that a Gecko chain can operate at 60MB/s to 120MB/s under heavy write-write contention and concurrent GC activity, whereas a conventional log-structured RAID-0 configuration over the same drives collapses to around 10MB/s during GC.

To tackle read-write contention caused by application reads, Gecko uses flash and RAM-based caching policies that leverage the unique structure of the logging chain. All new writes to the tail drive in the chain are first cached in RAM, and then lazily moved to an SSD cache dedicated to the tail drive. As a result, reads on recently written data on the tail drive are served by the RAM cache, and reads on older data on the tail drive are served by the SSD tail cache. This caching design has two important properties. First, it is tail-specific: it prevents application reads from reaching the tail drive and randomizing its workload, thus allowing writes to proceed sequentially without interference from reads. Based on our analysis of server block-level traces, we found that a RAM cache of 2GB and an SSD cache of 32GB was sufficient to absorb over 86% of reads directed at the 512GB tail drive of a Gecko chain for all the workload combinations we tried. Second, it's two-tier structure allows overwrites to be coalesced in RAM before they reach the SSD cache; as we show in our evaluation, this can prolong the lifetime of the SSD by 2X to 8X compared to a conventional caching design.

Chained logging has other benefits. Eliminating read-write contention has the side-effect that writes no longer slow down reads. As a result, chained logs can exhibit higher read throughput for many workloads compared to conventional RAID variants, since reads are served by either the tail cache or the body of the log and consequently do not have to contend with write traffic. Chained logging can also be used to save power: when mirrored drives are chained together, half the disks in the body of the log can be safely switched off since they do not receive any writes. This lowers the read throughput of the log, but does not compromise fault-tolerance.

Importantly, Gecko is a log-structured block device rather than a filesystem; as a result, any filesystem or database can execute over it without modification. Historically, the difficulty of persistently maintaining metadata under the block layer has outweighed the benefits of block-level logging, forcing such designs to incur metadata seeks on disk or restricting them to expensive enterprise storage solutions that can afford battery-backed RAM or other forms of NVRAM [43, 48, 141, 111, 83]. Gecko is the first system to use a commodity MLC SSD to store metadata for a log-structured disk array; accordingly, it uses a new metadata scheme carefully designed to exploit the access characteristics of flash as well as conserve its lifetime.

This chapter makes the following contributions. First, we propose the novel technique of chained logging, which provides the benefits of log-structured storage (obliviousness to write-write contention) without suffering from its drawbacks (susceptibility to read-write contention). Second, we describe the design of a block storage device called Gecko that implements chained logging, focusing on how the system utilizes inexpensive commodity flash for caching and persistence over the chained log structure. Third, we evaluate a software implementation of Gecko, showing that chained logging provides high, stable write throughput during GC activity, in contrast to log-structured RAID-0; it effectively prevents reads from impacting write throughput by using a tail-specific cache; and it outperforms log-structured RAID-0 in terms of both read and write performance on real workloads. Collectively, all these contributions lead to performance isolation in cloud storage servers.

3.1 Design

Gecko implements the abstraction of a block device, supporting reads and writes to a linear address space of fixed-size sectors. Underneath, this address space is implemented over a chained log structure, in which a single logical log is chained or concatenated across multiple drives such that the tail of the log and its body are on different drives. A new write to a sector in the address space is sent to the tail of the log; if it's an overwrite, the previous entry in the log for that sector is invalidated or trimmed. As the body of the log gets fragmented due to such overwrites on the address space, it is cleaned so that the freed space can be reused; importantly, this GC activity incurs reads on the body of the chained log, which do not interfere with first-class writes occurring at the tail drive of the log.

We first present the simplest possible instantiation of chained logging in Gecko, and then describe more sophisticated features. Gecko is implemented as a block device driver, occupying the same slot in the OS stack as software RAID; as with RAID, it can also be implemented in the form of a hardware controller. Gecko maintains an in-memory map (implemented as a simple array) from logical sectors on the supported address space to physical locations on the drives composing the array. In addition, it maintains an inverse map (also a simple array) to find the logical sector that a physical location stores; a special 'blank' value is used to indicate that the physical location does not contain valid data. Also, Gecko maintains two counters – one for the tail of the log and one for the head – each of which indexes into the total physical space available on the disk array.

When the application issues a read on a logical sector in the address space, the primary map is consulted to determine the corresponding physical location. When the application writes to a logical sector, the tail counter is checked and a write I/O is issued to the corresponding physical location on the tail drive. Both the primary map and the inverse map are then updated to reflect the linkage between the logical sector and the physical location, and the tail counter is incremented.

In the default form of GC supported by Gecko, data is constantly moved from the head of the chained log to its tail in order to reclaim space; we call this 'move-to-tail' GC. A cleaning process examines the next physical entry at the head of the log, checks if it is occupied by consulting the inverse map, and if so re-appends it to the tail of the log. It then increments the head and (if the entry was moved) the tail counter.

The basic system described thus far provides the main benefit of log chains – logging without interference from GC reads – but suffers from other problems. It does not offer tolerance to power failures or to disk failures. While GC writes do not drastically affect first-class writes, they do occur on the same drive as application writes and hence reduce write throughput to some extent. Further, the system is susceptible to contention between application reads and writes: reads to recently written data will go to the tail disk and disrupt first-class writes. Below, we discuss solutions to address these concerns.

3.1.1 Metadata

The total amount of metadata required by Gecko can easily fit into RAM on modern machines; to support a mirrored 4TB address space of 4KB sectors (i.e., 1 billion sectors) on an 16TB array, we need 4GB for the primary map (1 billion 4-byte entries), 8GB for the inverse map (2 billion 4-byte entries) and two 4-byte counters. However, a RAM-based solution poses the obvious problem of persistence: how do we recover the state of the Gecko address space from power failures?

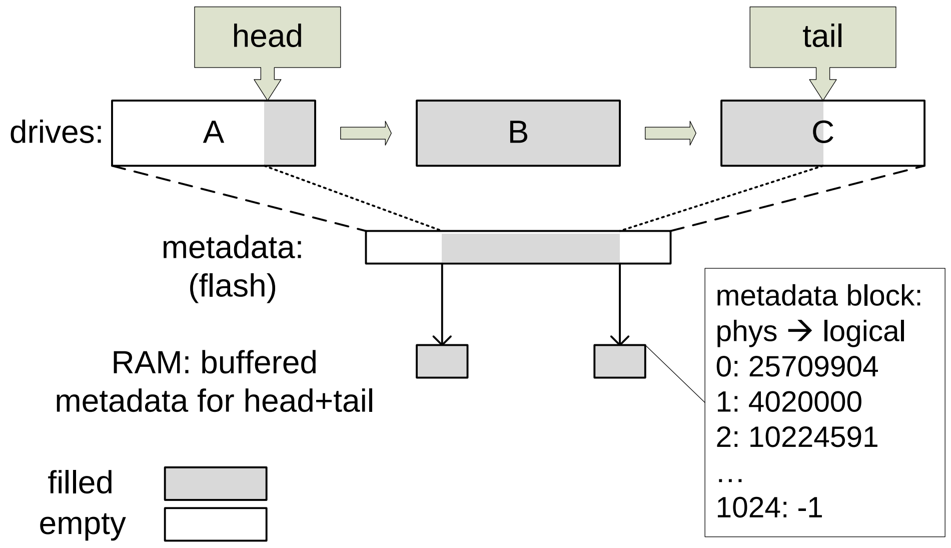


Figure 3.2: Metadata persistence in Gecko: mapping from physical to logical addresses is stored on flash, with actively modified head and tail metadata buffered in RAM.

One possibility is to store some part of the metadata on an SSD. An obvious candidate is the primary map, which is sufficient to reconstruct both the inverse map and the tail / head counters. Random reads on SSDs are fast enough (at roughly 200 microseconds) to exist comfortably in the critical path of a Gecko read. However, the primary map has very little update locality; a series of Gecko writes can in the worst case be distributed evenly across the entire logical address space. As a result, the metadata SSD is subjected to a workload of random 4-byte writes, which can wear it out very quickly.

Instead, Gecko provides persistence across power failures by storing the inverse map on an SSD, as shown in Figure 3.2. Each 4KB page on the SSD stores 1024 entries in the physical-to-logical map; we call this a metadata block. Accordingly, the larger log on the address space of the disk array is reflected at much smaller scale (a factor of 1K smaller) on the address space of the SSD. The *i*th 4-byte entry on the SSD is the logical address stored in the *i*th physical sector on the disk array. On a brand-new Gecko deployment, each such 4-byte metadata entry on the SSD is set to the 'blank' value, indicating that no valid data exists at that physical location on the array.

Gecko buffers a small number of metadata pages (in the simplest case, just one page) corresponding to the tail of the log in RAM; accordingly, as first-class writes are issued on the logical address space, these buffered metadata pages are modified in-memory. The metadata pages are flushed to the SSD when all entries in them have been updated, with the important condition that these flushes occur in strict sequential logging order. Correspondingly, Gecko also buffers the metadata pages at the head of the log during GC, which updates metadata entries to point to the 'blank' value. As a result of the flush-in-order condition, at any moment in time the SSD consists of two contiguous segments: one containing 'blank' entries and one with non-'blank' entries. As a result, on recovery from power failure, it is a simple task to reconstruct not only the primary map but also the head and tail counters, since they are simply the beginning and end of the contiguous non-'blank' segment.

The metadata buffering scheme described above avoids small random writes to the SSD due to the perfect update locality of the inverse map. However, it does introduce a window of vulnerability; all buffered metadata is lost on a power failure. A useful property of Gecko's log-structured design is that any such data loss is confined to a recent suffix of the log; in other words, the logical drive supported by Gecko simply reverts to an earlier (but consistent) state. If the application does want to guarantee durability of data, it can issue a 'sync' command to the Gecko block device, which causes Gecko to flush its current metadata page ahead of time to the SSD (and do an overwrite subsequently when the rest of the metadata page is updated). Alternatively, if Gecko is implemented as a hardware controller, battery-backed RAM or supercapacitors can be used to store the metadata pages being actively modified.

Under normal operation, this solution imposes a gentle, sequential workload on the SSD. The SSD only sees two 4KB page writes (one to change the entry from 'blank' to a valid location, and another to change it back during GC) for every 1024 4KB writes to the Gecko array. One of these writes can be avoided if the SSD supports a persistent trim command [92], since metadata blocks at the head can be trimmed instead of changed back to 'blank'. In the example above of a 16TB disk array with a mirrored 4TB address space, an 8GB SSD with 10K erase cycles (which should cost somewhere between $8 and $16 at current flash prices) should be able to support 10K times 8TB of writes, or 80PB of writes.

3.1.2 Caching

In Gecko, the role of caching is multi-fold: to reduce read latencies to data, but also to prevent application reads from interfering with writes (read-write contention). In conventional storage designs, it is difficult to predict which data to cache in order to minimize read-write contention. In contrast, eliminating read-write contention in Gecko is simply a matter of caching the data on the tail drive in the system, thus avoiding any disruption to the write throughput of the array.

To do so, Gecko uses a combination of RAM and an SSD (this can be a separate volume created on the same SSD used for storing metadata, or a separate SSD). When data is first written to a Gecko volume, it is sent to the tail drive and simultaneously cached in RAM. As a result, if the data is read back immediately, it can be served from RAM without disturbing sequentiality of the tail drive. As the tail drive and RAM cache continue to accept new data, older data is evicted from the RAM cache to the SSD cache in simple FIFO order (taking overwrites on the Gecko logical address space into account), and the SSD cache in turn uses an LRU-based eviction policy.

This simple caching scheme also prolongs the lifetime of the SSD cache by coalescing overwrites in the RAM cache. It is partly inspired by the technique of using a hard disk as a write cache for an SSD [124], and similarly extends the lifetime of the SSD by 2X to 8X.

Additionally, Gecko can optionally use RAM and SSD (again, another volume on the same SSD or a different drive) as a read cache for the body of the log, with the goal of improving read performance on the body of the log. In the rest of the chapter, we use the term ‘SSD cache’ to refer to the tail cache, unless explicitly specified otherwise.

3.1.3 Smarter Cleaning

Thus far, we have described the system as using move-to-tail GC, a simple cleaning scheme where data is moved in strict log order from the head of the log to its tail. While this scheme ensures that GC reads do not interfere with write throughput, GC writes do impact first-class writes to some extent. In particular, GC writes in move-to-tail GC do not disrupt the sequentiality of the tail drive, but instead take up a proportion of the sequential bandwidth of the drive; in the worst case where every element in the log is valid and has to be re-appended, this proportion can be as high as 50%, since every first-class write is accompanied by a single GC write.

To prevent GC writes from interfering with first-class writes, Gecko supports a more sophisticated form of GC called 'compact-in-body'. The key observation in compact-in-body is that any valid entry in the body of the log can be moved to any other position that succeeds it in the log without impacting correctness. Accordingly, instead of moving data from the head to the tail, we move it from the head to empty positions in the body of the log.

The cleaning process for compact-in-body GC is very similar to that of move-to-tail GC. It examines the next physical entry at the head of the log, checks if it is occupied by consulting the inverse physical-to-logical map, and if so, finds a free position in the body of the log between the current head and current tail. It then increments the head counter but leaves the tail counter alone (unless no free positions were found in the body of the log, forcing the update to go to the tail). Finding a free position requires the cleaning process to periodically scan ahead on the inverse map and create a free list of positions. These scans occur on the metadata SSD rather than the disk array and hence do not impact read throughput on the body of the log.

Compact-in-body has the significant benefit compared to move-to-tail that GC activity is now completely independent of first-class writes. It creates space at the head of the log by moving data to the body of the log rather than its tail, and hence does not use up a proportion of the write bandwidth of the tail drive. In addition, it requires no changes to the metadata or caching schemes described above.

However, as described, compact-in-body does have one major disadvantage; it randomizes the workload seen by the metadata SSD, since we are moving data from the head to free positions in the log, which could be randomly distributed. In practice, the difference in write bandwidth of a Gecko chain running move-to-tail GC versus compact-in-body GC is at most a factor of two, since move-to-tail GC uses up 50% of the tail drive's write bandwidth in the worst case whereas compact-in-body does not use any. Accordingly, we provide users the option of using either form of GC, depending on whether they want to maximize write bandwidth or minimize SSD wear.

3.1.4 Discussion

Chain Length: As mentioned previously, chained logging is based on the premise that the sequential write throughput of a single, uncontended drive is preferable to the overall throughput of multiple, contention-hit drives. This argument obviously does not scale to a large number of drives; beyond a certain array size, the random write throughput of the entire array will exceed the sequential throughput of a single drive. The shorter the length of the chain, the more likely it is that chained logging will outperform conventional RAID-0 over the same number of drives.

However, longer chains have other benefits, such as the improved read throughput that results from having multiple disk heads service the body of the log. Another reason for longer chains is that it allows capacity to be added to the physical log. This capacity can be used to either extend the size of the supported address space, or to lower garbage collection stress on the same address space. In practice, we find that chains of two to four drives provide a balance between write throughput, read throughput and capacity.

Multiple Chains: We expect multiple Gecko chains to be deployed on a single system; for example, a 32-core system with 24 disks might have four mirrored chains of length 3, each serving a set of 8 cores. A single metadata SSD can be shared by all the chains, since the metadata has a simple one-to-one mapping to the physical address space of the entire system. A single cache SSD can be partitioned across chains, with each chain using a 32GB cache.

On a large system with multiple chains, each chain can be extended or shortened on the fly by moving drives to and from other chains, as the occupancy (and consequently, GC demands) of the supported address space and the read-/write ratio of the workload change over time. Read-intensive workloads require more disks to be dedicated to the body of the chain.

System Cost: The design described thus far requires: an SSD read cache for the tail, an SSD read cache for the body, a metadata SSD, and a few GB of RAM per chain. Consider an array of 30 512GB drives (15TB in total), organized into 5 mirrored chains of length 3. Based on our experience with Gecko, each such chain requires 2GB of RAM, 32GB of flash for the tail cache, 32GB of flash for the body cache, and 1.5GB of flash for metadata; the total for 5 chains is 10GB RAM and around 340GB of flash. At current RAM and flash prices, this amounts to less than $500, a reasonably small fraction of the total cost for such a system.

Mirroring: As described earlier, a Gecko chain can consist of mirrored drive pairs. Mirroring is very simple to implement; since the drives are paired deterministically and kept perfectly synchronized, none of the Gecko data structures need to be modified. Some benefits of mirroring are obvious, such as fault tolerance against drive failures and higher read throughput. A more subtle point is that Gecko facilitates power saving when used over mirrored drives. Since writes in chained logs only happen at the tail, drives in the body of the log can be powered down as long as one mirror stays awake to serve reads. In a chain consisting of three mirrored pairs, two drives (or a third of the array) can be powered down without affecting data availability. With longer chains, a larger fraction of the array can be powered down.

Additionally, Gecko can potentially perform decoupled GC on mirrors, allowing one drive to serve first-class reads while cleaning the other drive. This complicates the metadata structures maintained by Gecko, both in RAM as well as the metadata SSD, since it needs to now maintain state for each drive separately. Due to the increased complexity of this option, we chose not to explore it further.

Striping: Gecko can also be easily combined with striping, simply by having each drive in the chain be a striped RAID-0 volume. This allows a single Gecko address space to scale to larger numbers of drives. One implication of striping is that the tail drive(s) now have much greater capacity and may require proportionally larger SSD caches to prevent reads from impacting them. Other RAID variants such as RAID-5 and RAID-6 can be layered in similar fashion under Gecko without any change to the system design.

3.2 Evaluation

We have implemented Gecko as a device driver in Linux that exposes a block device to applications. This device driver implements move-to-tail GC and a simplistic form of persistence involving checkpointing all metadata to an SSD every few minutes. In addition, we also implemented a user-space emulator to test the more involved aspects of Gecko, such as the metadata logging design for persistence described in Section 3.1.1, compact-in-body GC, and different caching policies. All our experiments were conducted on a system with a 12-core Intel Xeon processor, 24GB RAM, 15 10K RPM drives of 600GB each, and a single 120GB SSD.

Our main baseline for comparison is a conventional log layered over either RAID-0 or RAID-10 (which we call log-structured RAID-0 / RAID-10), comparable respectively to the non-mirrored and mirrored Gecko deployments. For instance, an array of six drives may be configured as a 3-drive Gecko chain, where each drive is mirrored; for this, the comparison point would be a log-structured RAID-10 volume with three stripes, each of which is mirrored. To implement this log-structured RAID design, we treat the entire array as a single RAID-0 or RAID-10 volume and then run a single-drive Gecko chain over it; this ensures that we use identical, optimized code bases for both Gecko and the baseline. When appropriate, we also report numbers on in-place (as opposed to log-structured) RAID-0, though most of our workloads have enough random I/O that in-place RAID-0 only offers a few MB/s and is not competitive.

Our evaluation focuses on three aspects of Gecko. First, we show that a Gecko chain implementing move-to-tail GC is capable at operating at high, stable write throughput even during periods of high GC activity under an adversarial workload, whereas the write throughput of log-structured RAID-0 drops drastically. This validates our claim that Gecko write throughput does not suffer from contention with GC reads. Second, we show that our RAM+SSD caching policies are capable of eliminating almost all first-class reads from the tail drive for a majority of tested workloads, while preserving the lifetime of the SSD cache. Thus, we show that Gecko write throughput does not suffer from contention between application reads. Finally, we play back real traces on a Gecko deployment and show that Gecko offers higher write throughput as well as higher read throughput compared to log-structured RAID-10.

3.2.1 Write Throughput with GC

To show that Gecko can sustain high write throughput despite concurrent GC, we ran a synthetic workload of random writes from multiple processes over the block address space exposed by the Gecko in-kernel implementation. In this experiment, we used a 2-drive, non-mirrored Gecko chain and a conventional log layered over 2-drive RAID-0. Midway through the workload, we turned on GC for Gecko and measured the resulting drop in total and application throughput. For the log-structured RAID-0, we triggered GC for the same time period as Gecko. Figure 3.3 (Top) shows Gecko throughput for different trim patterns in the body of the log; e.g., a trim pattern with 50% valid data has half the blocks in the body of the log marked as invalid, while the other half is valid and has to be moved by GC to the tail.

As shown in the figure, Gecko throughput remains high and steady during

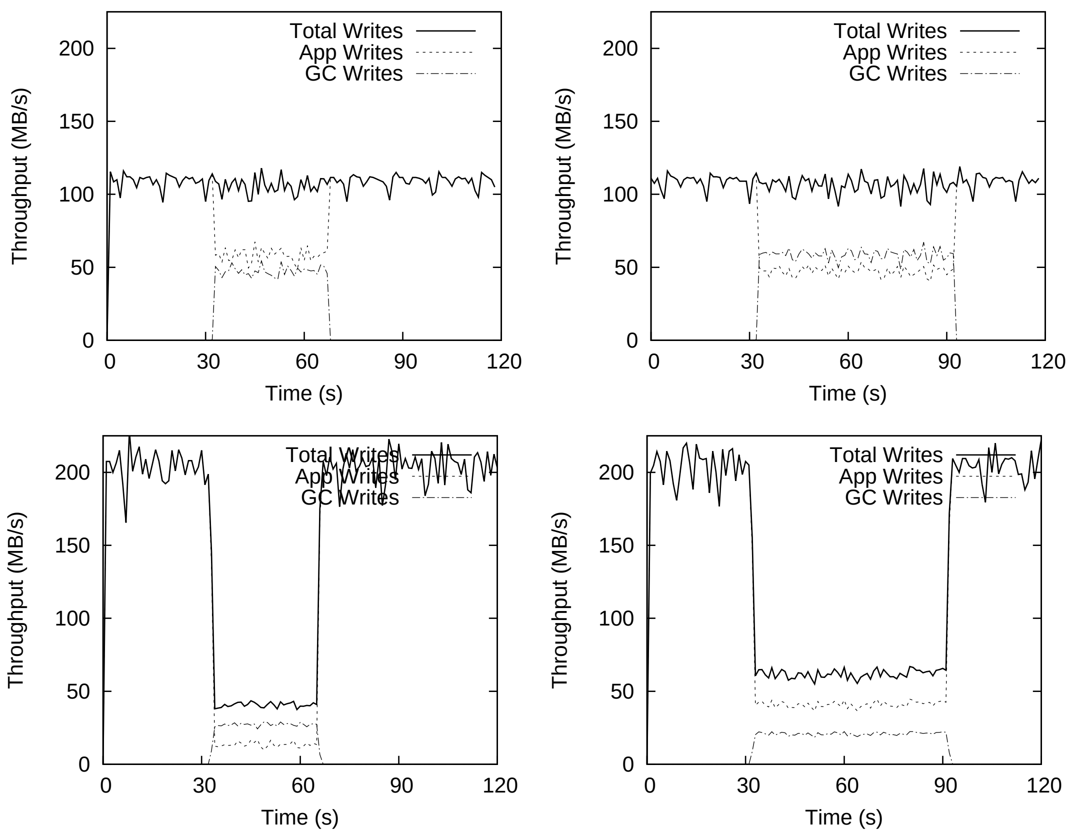


Figure 3.3: Gecko (Top) offers steady, high application throughput (60MB/s or 15K IOPS) for a random write workload during GC with a 50% trim pattern (Left) and a 0% trim pattern (Right). Log-structured RAID-0 (Bottom) suffers application throughput collapse to 10MB/s for 50% trims (Left) and provides 40MB/s for 0% trims.

GC activity, while application throughput drops proportionally to accommodate GC writes. We trigger GC to clear a fixed amount of physical space in the log; as a result, the 50% trim pattern (Top Left) has a GC valley that is approximately half as wide as that of the 0% trim pattern (Top, Right), since it moves exactly half the amount of data. The two different trim patterns on the body of the log do not impact Gecko write throughput in any way, showing that the strategy of decoupling the tail of the log from its body succeeds in shielding write throughput from GC read activity.

In contrast, the log-structured RAID-0 in Figure 3.3 (Bottom) performs very poorly when GC is turned on for the 50% trim pattern; throughput collapses drastically to the 10MB/s mark. Counter-intuitively, it performs better for 0% trim pattern; even though more data has to be moved in this pattern, the GC reads to the drive are sequential, causing less disruption to the write throughput of the array. An important point is that Gecko cleans 2X to 3X the physical space compare to log-structured RAID-0 in the same time period: the top Gecko graphs show almost 4GB of log space being reclaimed while the bottom log-structured RAID-0 graphs show reclamation of approximately 1.5 GB of log space in a 40 second (Left) and 60 second (Right) period.

One point to note is that Gecko does suffer from a drop in application throughput, or goodput, due to GC. In the worst case where all data is valid and has to be moved (shown in the top right figure), application throughput can drop by exactly half. This represents a lower bound on application throughput, since in the worst case every new write requires a single GC write to clear up space in the physical log. Accordingly, Gecko application throughput is bounded between 60MB/s (half the sequential bandwidth of a single drive) and 120MB/s (the full sequential bandwidth of a drive), with the exact performance depending on the size of the supported logical address space, as well as the pattern of overwrites observed by it. Not shown in the figure is in-place RAID-0, which provided only a few MB/s under this random writes workload, as expected.

Next, we ran the Gecko emulator in compact-in-body mode as well as move-to-tail mode for a random write workload with a 50% trim pattern. Figure 3.4 shows that compact-in-body GC allows application writes to proceed at the full

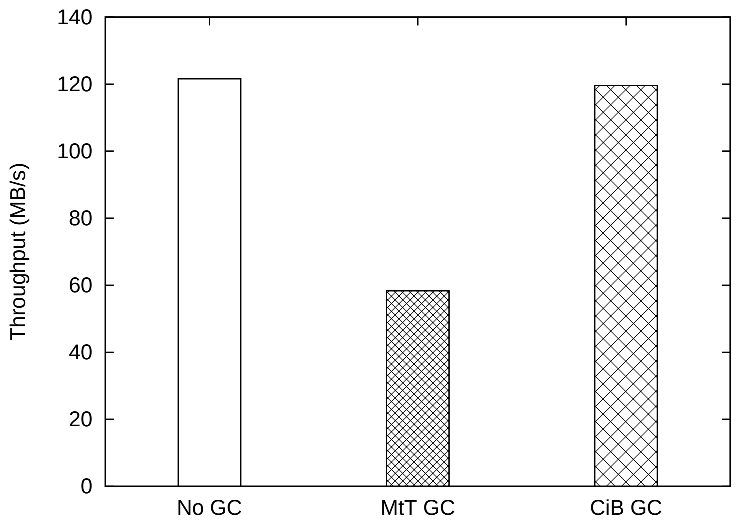


Figure 3.4: With compact-in-body GC (CiB), a log chain of length 2 achieves 120MB/s application throughput on random writes with concurrent GC on 50% trims.

sequential speed of the tail drive during GC activity. As discussed previously, this performance benefit comes at the cost of erase cycles on the metadata SSD; accordingly, we do not explore compact-in-body GC further.

3.2.2 Caching the Tail

Having established that Gecko provides high write throughput in the presence of GC activity, we now focus on contention between first-class reads and writes. We show that Gecko can effectively cache data on the tail drive in order to prevent contention between first-class reads and writes. In these experiments, we use block-level traces taken from the SNIA repository [20]; specifically, we use the Microsoft Enterprise, Microsoft Production Server and MSR Cambridge trace sets. Running these traces directly over Gecko is unrealistic, since they were collected on non-virtualized systems. Instead, we run workload combina-

Raw Trace	*GB of Writes*
A. DevDivRelease	176.1
B. Exchange	459.6
C. LiveMapsBE	558.2
D. prxy	778.6
E. src1	883.7
F. proj	342.2
G. MSNFS	102.3
H. prn	76.8
I. usr	95.7

Combination 0 – 7: any 8 from {A,...,I}
Combination 8 – 20: any 4 from {A,...,E}

Table 3.1: Workload combinations: from 9 raw traces, we can compose 8 8-trace combinations and 13 4-trace combinations that write at least 512GB of data.

tions by interleaving I/Os from sets of either 4 or 8 traces, to emulate a system running different workloads within separate virtual machines. We play each trace within its own virtual address space and concatenate each of these together to obtain a single logical address space.

To study the effectiveness of Gecko's tail caching, we ran multiple such workload combinations over our user-space Gecko emulator, starting with an empty tail drive. We then measured the hit rate of Gecko's hybrid cache consisting of 2GB of RAM and a 32GB SSD. Recall that new writes in Gecko go to the tail drive and are simultaneously cached in RAM, and subsequently evicted from RAM to the SSD. A cache hit is when data that resides on the tail drive is also found in either RAM or the SSD; conversely, a cache miss occurs when data that resides in the tail drive is not found in RAM or the SSD, necessitating a read from the tail drive. Note that any read to data that does not exist on the tail drive is ignored in this particular experiment, since it will be serviced by the body of the log without causing read-write contention.

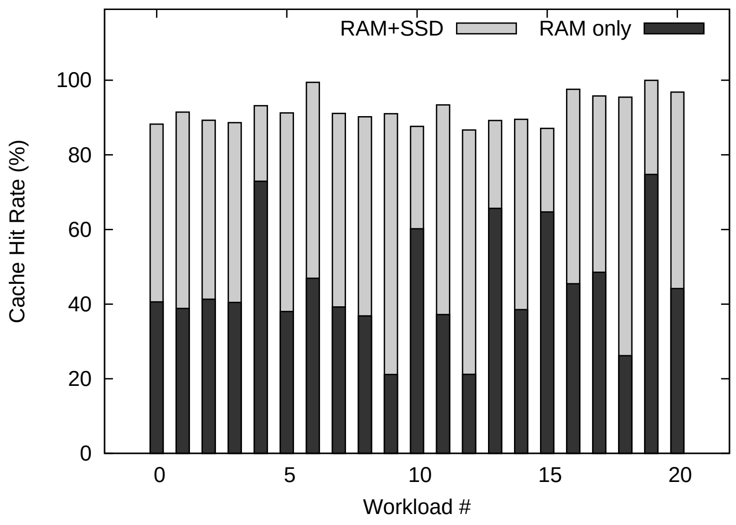


Figure 3.5: Effectiveness of tail caching on different workload combinations with a 2GB RAM + 32GB SSD cache. The hit rate is over 86% for all 21 combinations, over 90% for 13, and over 95% for 6.

To avoid overstating cache hit rates, we needed each workload combination to write at least 512GB (i.e., the size of the tail drive); as we show later, cache hit rates are very high as we start writing to the tail drive, but drop as it fills up. From the 21 SNIA traces, we found 8 8-trace combinations that lasted at least 512GB (which we number 0 to 7), and 13 4-trace combinations that lasted at least 512GB (which we number 8 to 20), for a total of 21 workload combinations of at least 512GB each. These workload combinations used 9 of the 21 raw SNIA traces, as shown in Table 3.1; the remaining 12 raw traces did not have enough writes to be useful for this caching analysis.

Figure 3.5 shows cache hit rates – for just the 2GB RAM cache as well as for the combined 2GB+32GB RAM+SSD cache – for these 21 workload combinations, measured over the time that the 512GB tail drive is filled. The hit rate is over 86% for all tested combinations, over 90% for 13 of them, and over 95% for 6 of them. This graph validates a key assumption of Gecko: the tail drive

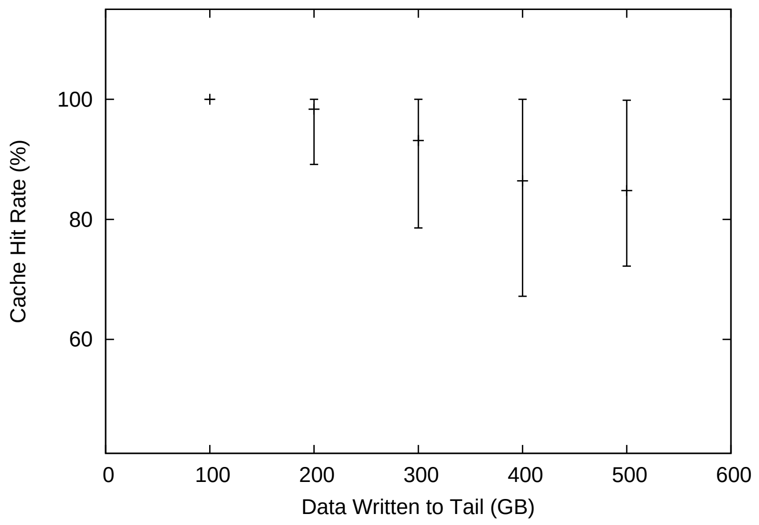


Figure 3.6: Average, min and max hit rates of tail caching across workload combinations as the tail drive fills up.

of a chained log can be cached effectively, preventing application reads from disrupting the sequential write throughput of the log.

Next, we measured how the cache hit rate changes over time as the tail drive fills up. Figure 3.6 shows the average hit rate across the 21 workload combinations for the RAM+SSD cache, in each consecutive 100GB interval on the tail drive (the error bars denote the min and the max across the workload combinations). The hit rate is extremely high for the first 100GB of data, as the total amount of data on the tail drive is not much bigger than the cache. As expected, the hit rate dips as more data is stored on the tail. Note that Figure 3.5 previously showed the cumulative hit rate over 512GB of writes, whereas this figure shows the hit rate for each 100GB interval separately.

We claimed earlier that Gecko's two-tier RAM+SSD caching scheme could prolong the lifetime of the SSD compared to an SSD-only cache by coalescing overwrites in RAM. Following the methodology in [124], we calculate the life-

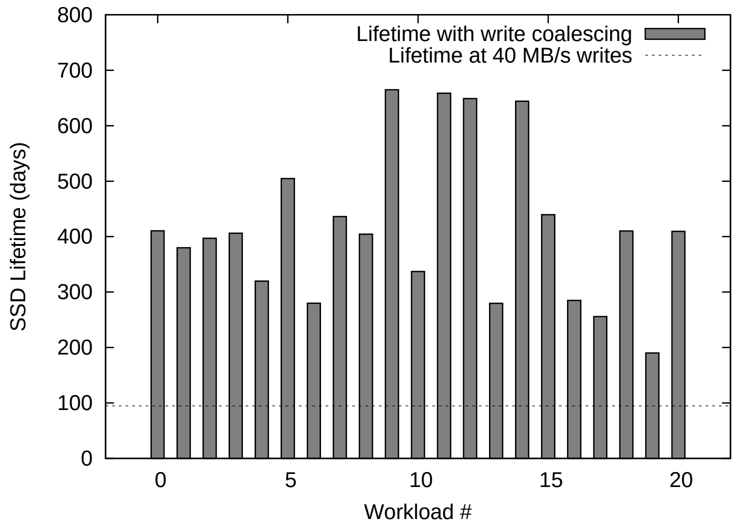


Figure 3.7: Gecko's hybrid caching scheme for its tail drives increases the lifetime of the SSD read cache by at least 2X for all 21 workload combinations, and by more than 4X for 13 combinations.

time of an SSD by assuming a one-to-one ratio between page writes sent to the SSD and erase cycles used per page, and assuming that the SSD supports 10,000 block erase cycles. Under these assumptions, a constant 40MB/s workload will wear out a 32GB SSD in approximately 3 months; accordingly, this would be the lifetime of a conventional SSD-based write or read cache if the system were written to continuously at 40MB/s.

By using a RAM+SSD read cache and coalescing overwrites in RAM, we decrease the number of writes going to the SSD by a factor of 2X to 8X for different workload combinations. In Figure 3.7, we plot the number of days the SSD lasts with write coalescing, under the assumptions previously stated. For some workload combinations, we are able to stretch out the SSD lifetime to over two years even at this high 40MB/s update rate; for all of them, we at least double the SSD lifetime. A simple linear relationship exists between these numbers and the average data rate of the system; at 20MB/s, for instance, the SSD will last twice as long. Alternatively, we can use larger capacity SSDs to extend the SSD replacement cycle: e.g. with a 64GB SSD, the cycle can double if one uses the first half until it wears out and then uses the other half.

3.2.3 Gecko Performance for Real Workloads

To show that effective tail-caching results in better performance, we played two 8-trace combinations – specifically, the ones with the highest and lowest cache hit rates – over the Gecko implementation. In this experiment, we played each trace combination as fast as possible, issuing the appropriate I/Os to either the SSD cache or disk. We used a single outstanding I/O queue of size 24 for each trace in the combination, shared by reads and writes.

For Gecko, we used a 3-drive mirrored chain with a 2GB RAM + 32GB SSD tail cache and a separate 32GB SSD cache for the body of the log. For comparison, we used a conventional log over a 6-drive RAID-10 volume with a single unified cache for the entire array, consisting of 2GB RAM and 64GB SSD.

In the experiment, we played the trace combination forward until 200GB of the tail was filled before taking measurements, to ensure that we obtained average caching performance on the tail. Reads on logical addresses that had not yet been written were directed to random locations on the body of the log.

Figure 3.8 shows the total read plus write throughput of the system as well as just write throughput over a 120 second period. On top we show the highly cacheable workload combination; on bottom we show the less cacheable one. On the left we show Gecko performance, while on the right we show the perfor-

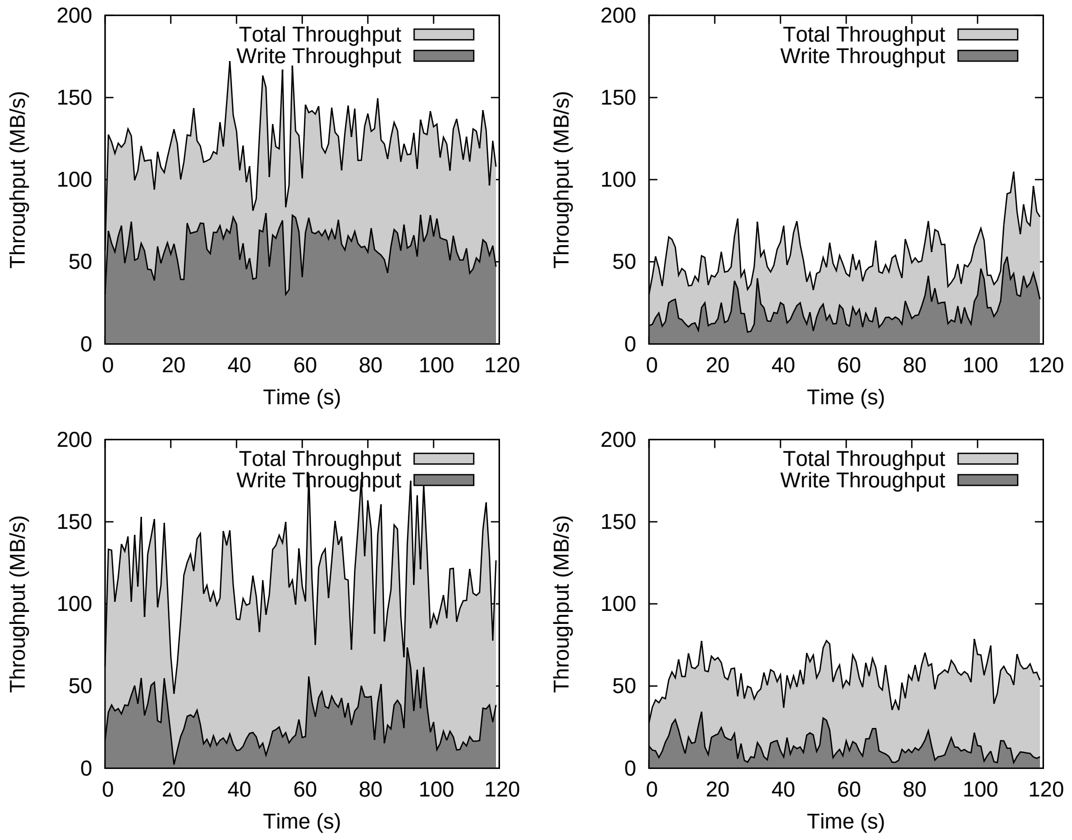


Figure 3.8: Gecko (Left) offers 2X to 3X higher throughput than log-structured RAID-10 (Right) on a highly cacheable (Top) and less cacheable (Bottom) workload combination for writes as well as reads.

mance of log-structured RAID-10. No GC activity was triggered concurrently, in order to isolate the impact of first-class reads on performance.

At a basic level, it's clear that Gecko outperforms log-structured RAID-10 by 2X to 3X on both workloads. Gecko offers lower write performance than expected, since write throughput is not pegged at 120MB/s; this is an artefact of our trace playback process, since our fixed-size window of I/Os ends up clogged with the slower reads on the body of the log, preventing new writes from being issued. Surprisingly, Gecko offers much better read performance than log-structured RAID-10, again by a factor of 2X to 3X; in effect, separating reads and writes has a positive effect on reads, which do not have to contend with write traffic anymore. Especially, all fresh reads that are not cached from recent writes contend with writes in both cache and disks for log-structured RAID-10 and this significantly lowers the throughput for both reads and writes. An interesting point is that both workloads are highly cacheable for reads; our classification of these workloads as highly cacheable and less cacheable was based on the cacheability of the tail drive, which does not seem to correlate to the cacheability of the body.

To test the performance under GC, we triggered move-to-tail GC in the same setup as in Figure 3.8 with 700GB of data pre-filled. Approximately 75% of data was trimmed for both workloads and the average total throughputs of Gecko dropped to 65MB/s and 62MB/s for highly and less cacheable workloads respectively due to contention between first-class reads and GC reads. However, Gecko still outperformed log-structured RAID-10 performing GC by over 2X to 3X. The average application throughputs of Gecko were 33MB/s and 27MB/s whereas those of log-structured RAID-10 were 13MB/s and 12MB/s for the respective workloads.

Finally, we plot the impact of chain length on throughput in Figure 3.9. We run the highly cacheable workload from the previous experiments on Gecko and log-structured RAID-0, measuring read and write throughput while increasing the number of drives used without GC. In Gecko, more drives in the array translates into more drives in the body of the chain, while for log-structured RAID-0 it provides more disks to stripe over. As the graph shows, a single Gecko chain outperforms log-structured RAID-0 for both reads and writes even on a 7-disk array. Essentially, two key principles in the Gecko design continue

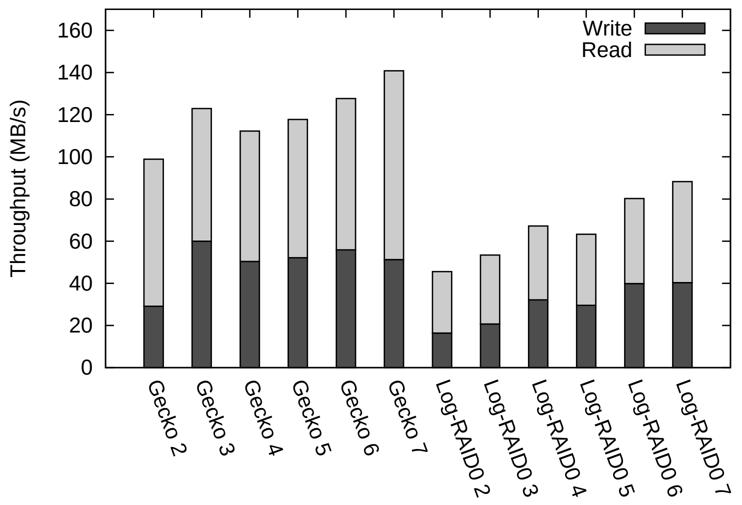


Figure 3.9: A Gecko chain outperforms log-structured RAID-0 even on 7 drives: one uncontended disk for writes is better than many contention-prone disks.

to hold even for long chains: a single uncontended disk arm is better for write performance than multiple contended disk arms; and segregating reads from writes enables better read performance.

3.3 Summary

In this chapter, we demonstrated how to resolve disk contention by using a chained logging design. A number of factors herald a resurgence of log-structured storage designs in cloud storage systems: the prevalence of many-core machines and the availability of flash-based read caches. Log-structured designs have the potential to be a panacea for storage contention in the cloud; however, they continue to be plagued by the cleaning-related performance issues that held back widespread deployment in the 1990s. Gecko attempts to solve this long-standing problem and provide performance isolation in cloud storage servers by separating the tail of the log from its body to isolate cleaning activity completely from application writes. A dedicated cache for the tail drive prevents reads from interfering with writes. Using these mechanisms, Gecko offers the benefits of a log-structured design without its drawbacks, presenting system designers with a storage system with improved performance isolation.

CHAPTER 4

TRANSACTIONAL ISOLATION SUPPORT FROM THE BLOCK LAYER

To support transactional isolation in a cloud storage server, the transactional functionality should be placed in a lower layer of the storage stack so that all applications running in the server can access the functionality with ease. Transactional isolation in a cloud storage server is becoming more important with the advent of multi-core machines and storage systems such as filesystems, key-value stores, graph stores and databases which are increasingly parallelized over dozens of cores. Such systems run directly or indirectly over the block layer but assume very little about its interface and semantics. As a result, each system implements complex code to layer high-level semantics such as transactional atomicity and transactional isolation over the simple block address space. Redundant implementations of transactional functionalities in the high layer of storage stacks suggest rethinking the storage stack design and pushing the transactional functionality down to the block layer.

In this chapter, we propose the abstraction of a transactional block store that provides isolation in addition to atomicity and durability in the cloud storage server. While multiple systems have implemented transactional atomicity within the block store [43, 48, 101, 17, 44], transactional isolation has traditionally been delegated to the storage system above the block store. A number of factors make isolation a prime candidate for demotion down the stack.

1) Isolation is *general*; since practically every storage system has to ensure safety in the face of concurrent accesses, an isolation mechanism implemented within the block layer is broadly useful.

2) Isolation is *hard*, especially for storage systems that need to integrate fine-

grained concurrency control with coarse-grained durability and atomicity mechanisms (e.g., see ARIES [87]); accordingly, it is better provided via a single, high-quality implementation within the block layer.

3) Block-level transactions allow storage systems to effortlessly provide end-user applications with transactions over high-level constructs such as files or key-value pairs.

4) Block-level transactions are oblivious to software boundaries at higher levels of the stack, and can seamlessly span multiple layers, libraries, threads, processes, and interfaces. For example, a single transaction can encapsulate an end application's accesses to an in-process key-value store, an in-kernel filesystem, and an out-of-process graph store.

5) Finally, multiversion concurrency control (MVCC) [38] provides superior performance and liveness in many cases but is particularly hard to implement for storage systems since it requires them to maintain multiversioned state; in contrast, many block stores (e.g., log-structured designs) are already internally multiversioned.

Block-level isolation is enabled and necessitated by several trends in block stores. Block stores are increasingly implemented via a combination of host-side software and device firmware [21, 7]; they incorporate multiple, heterogeneous physical devices under a single address space [124, 119]; they leverage new NVRAM technologies to store indirection metadata; and they provide sophisticated functionality such as virtualization [21, 126], tiering [21], deduplication and wear-leveling. Unfortunately, storage systems such as filesystems continue to assume minimum functionality from the block store, resulting in redundant, complex, and inefficient stacks where layers constantly tussle with each other [126]. A second trend that argues for pushing functionality from the filesystem to a lower layer is the increasing importance of alternative abstractions that can be implemented directly over block storage, such as graphs, key-value pairs [19], tables, caches [113], tracts [93], byte-addressable [32] and write-once [33] address spaces, etc.

To illustrate the viability and benefits of block-level isolation in a cloud storage server, we present Isotope, a transactional block store that provides isolation (with a choice of strict serializability or snapshot isolation) in addition to atomicity and durability. Isotope is implemented as an in-kernel software module running over commodity hardware, exposing a conventional block read/write interface augmented with *beginTX*/*endTX* IOCTLs to demarcate transactions. Transactions execute speculatively and are validated by Isotope on *endTX* by checking for conflicts. To minimize the possibility of conflict-related aborts, applications can provide information to Isotope about which sub-parts of each 4KB block are read or written, allowing Isotope to perform conflict detection at sub-block granularity.

Internally, Isotope uses an in-memory multiversion index over a persistent log to provide each transaction with a consistent, point-in-time snapshot of a block address space. Reads within a transaction execute against this snapshot, while writes are buffered in RAM by Isotope. When *endTX* is called, Isotope uses a new MVCC commit protocol to determine if the transaction commits or aborts. The commit/abort decision is a function of the timestamp-ordered stream of recently proposed transactions, as opposed to the multiversion index; as a result, the protocol supports arbitrarily fine-grained conflict detection without requiring a corresponding increase in the size of the index. When transactions commit, their buffered writes are flushed to the log, which is implemented on an array of physical drives [119], and reflected in the multiversion index. Importantly, aborted transactions do not result in any write I/O to persistent storage.

Storage systems built over Isotope are simple, stateless, shim layers that focus on mapping some variable-sized abstraction – such as files, tables, graphs, and key-value pairs – to a fixed-size block API. We describe several such systems in this chapter, including a key-value store based on a hashtable index, one based on a B-tree, and a POSIX user-space filesystem. These systems do not have to implement their own fine-grained locking for concurrency control and logging for failure atomicity. They can expose transactions to end applications without requiring any extra code. Storage systems that reside on different partitions of an Isotope volume can be composed with transactions into larger end applications.

Block-level isolation does have its limitations. Storage systems built over Isotope cannot share arbitrary, in-memory soft state such as read caches across transaction boundaries, since it is difficult to update such state atomically based on the outcome of a transaction. Instead, they rely on block-level caching in Isotope by providing hints about which blocks to cache. We found this approach well-suited for both the filesystem application (which cached inode blocks, indirection blocks and allocation maps) and the key-value stores (which cached their index data structures). In addition, information is invariably lost when functionality is implemented at a lower level of the stack: Isotope cannot leverage properties such as commutativity and idempotence while detecting conflicts.

This chapter makes the following contributions:

- We propose the abstraction of a fully transactional block store that provides isolation, atomicity and durability. While others have explored block-level transactional atomicity [43, 48, 101, 44], this is the first proposal for block-level transactional isolation.
- We realize this abstraction in a system called Isotope via a new MVCC protocol. We show that Isotope exploits sub-block concurrency in workloads to provide a high commit rate for transactions and high I/O throughput.
- We describe storage systems built using Isotope transactions – two key-value stores and a filesystem – and show that they are simple, fast, and robust, as well as composable via Isotope transactions into larger end applications.

4.1 The Isotope API

The basic Isotope API is shown in Figure 4.1: applications can use standard POSIX calls to issue reads and writes to 4KB blocks, bookended by *beginTX*/*endTX* calls. The *beginTX* call establishes a snapshot; all reads within the transaction are served from that snapshot. Writes within the transaction are speculative. Each transaction can view its own writes, but the writes are not made visible to other concurrent transactions until the transaction commits. The *endTX* call returns true if the transaction commits, and false otherwise. The *abortTX* allows the application to explicitly abort the transaction. The application can choose one of two isolation levels on startup: strict serializability or snapshot isolation.

The Isotope API implicitly associates transaction IDs with user-space

```
/*** Transaction API ***/
int beginTX();
int endTX();
int abortTX();
//POSIX read/write commands

/*** Optional API ***/
//release ongoing transaction and return handle
int releaseTX();

//take over a released transaction
int takeoverTX(int tx_handle);

//mark byte range accessed by last read/write
int mark_accessed(off_t blknum, int start, int size);

//request caching for blocks
int please_cache(off_t blknum);
```

Figure 4.1: The Isotope API.

threads, instead of augmenting each call signature in the API with an explicit transaction ID that the application supplies. We took this route to allow applications to use the existing, highly optimized POSIX calls to read and write data to the block store. The control API for starting, committing and aborting transactions is implemented via IOCTLs. To allow transactions to execute across different threads or processes, Isotope provides additional APIs via IOCTLs: *releaseTX* disconnects the association between the current thread and the transaction, and returns a temporary transaction handle. A different thread can call *takeoverTX* with this handle to associate itself with the transaction.

Isotope exposes two other optional calls via IOCTLs. After reading or writing a 4KB block within a transaction, applications can call *mark_accessed* to explicitly specify the accessed byte range within the block. This information is key for fine-grained conflict detection; for example, a filesystem might mark a single inode within an inode block, or a single byte within a data allocation bitmap. Note that this information cannot be inferred implicitly by comparing the old and new values of the 4KB block; the application might have overwritten parts of the block without changing any bits. The second optional call is *please_cache*, which lets the application request Isotope to cache specific blocks in RAM; we discuss this call in detail later in the chapter. Figure 4.2 shows a snippet of application code that uses the Isotope API (the *setattr* function from a filesystem).

If a read or write is issued outside a transaction, it is treated as a singleton transaction. Singleton reads see all prior committed data since they access the latest snapshot of the system. Singleton writes always commit and are immediately durable. In effect, Isotope behaves like a conventional block device if the reads and writes issued to it are all non-transactional. In addition, Isotope can preemptively abort transactions to avoid buggy or malicious applications from hoarding resources within the storage subsystem. When a transaction is preemptively aborted, any reads, writes, or control calls issued within it will return error codes, except for *endTX*, which will return false, and *abortTX*.

Transactions can be nested – i.e., a *beginTX*/*endTX* pair can have other pairs nested within it – with the simple semantics that the internal transactions are ignored. A nested *beginTX* does not establish a new snapshot, and a nested *endTX* always succeeds without changing the persistent state of the system. A nested *abortTX* causes any further activity in the transaction to return error codes until all the enclosing *abortTX*/*endTX* have been called. This behavior is important for allowing storage systems to expose transactions to end-user applications. In the example of the filesystem, if an end-user application invokes *beginTX* (ei-

```
isofs_inode_num ino;
unsigned char *buf;

//allocate buf, set ino to parameter
...
int blknum = inode_to_block(ino);

txbegin:
beginTX();

if(!read(blknum, buf)){
        abortTX();
        return EIO;
}
mark_accessed(blknum, off, sizeof(inode));

//update attributes
...

if(!write(blknum, buf)){
        abortTX();
        return EIO;
}
mark_accessed(blknum, off, sizeof(inode));

if(!endTX())
        goto txbegin;
```

Figure 4.2: Example application: *setattr* code for a filesystem built over Isotope.

ther directly on Isotope or through a filesystem-provided API) before calling the *setattr* function in Figure 4.2 multiple times, the internal transactions within each *setattr* call are ignored and the entire ensemble of operations will commit or abort atomically.

4.1.1 Composability

As stated earlier, a primary benefit of a transactional block store is its obliviousness to the structure of the software stack running above it, which can range from a single-threaded application to a composition of multi-threaded application code, library storage systems, out-of-process daemons and kernel modules. The Isotope API is designed to allow block-level transactions to span arbitrary compositions of different types of software modules. We describe some of these composition patterns in the context of a simple photo storage application called ImgStore, which stores photos and their associated metadata in a key-value store.

In the simplest case, ImgStore can store images and various kinds of metadata as key-value pairs in IsoHT, which in turn is built over a Isotope volume using transactions. Here, a single transaction-oblivious application (ImgStore) runs over a single transaction-aware library-based storage system (IsoHT).

Cross-Layer: ImgStore may want to atomically update multiple key-value pairs in IsoHT; for example, when a user is tagged in a photo, ImgStore may want to update a photo-to-user mapping as well as a user-to-photo mapping, stored under two different keys. To do so, ImgStore can encapsulate calls to IsoHT within Isotope *beginTX*/*endTX* calls, leveraging nested transactions.

Cross-Thread: In the simplest case, ImgStore executes each transaction within a single thread. However, if ImgStore is built using an event-driven library that requires transactions to execute across different threads, it can use the *releaseTX*/*takeoverTX* calls.

Cross-Library: ImgStore may find that IsoHT works well for certain kinds of accesses (e.g., retrieving a specific image), but not for others such as range queries (e.g., finding photos taken between March 4 and May 10, 2015). Accordingly, it may want to spread its state across two different library key-value stores, one based on a hashtable (IsoHT) and another on a B-tree (IsoBT) for efficient range queries. When a photo is added to the system, ImgStore can transactionally call *put* operations on both stores. This requires the key-value stores to run over different partitions on the same Isotope volume.

Cross-Process: For various reasons, ImgStore may want to run IsoHT in a separate process and access it via an IPC mechanism; for example, to share it with other applications on the same machine, or to isolate failures in different codebases. To do so, ImgStore has to call *releaseTX* and pass the returned transaction handle via IPC to IsoHT, which then calls *takeoverTX*. This requires IsoHT to expose a transaction-aware IPC interface for calls that occur within a transactional context.

4.2 Design and Implementation

Figure 4.3 shows the major components of the Isotope design. Isotope internally implements an in-memory multiversion index (*B* in the figure) over a persistent log (*E*). Versioning is provided by a timestamp counter (*A*) which determines the snapshot seen by a transaction as well as its commit timestamp. This commit timestamp is used by a decision algorithm (*D*) to determine if the transaction commits or not. Writes issued within a transaction are buffered (*C*) during its execution, and flushed to the log if the transaction commits. We now describe the interaction of these components.

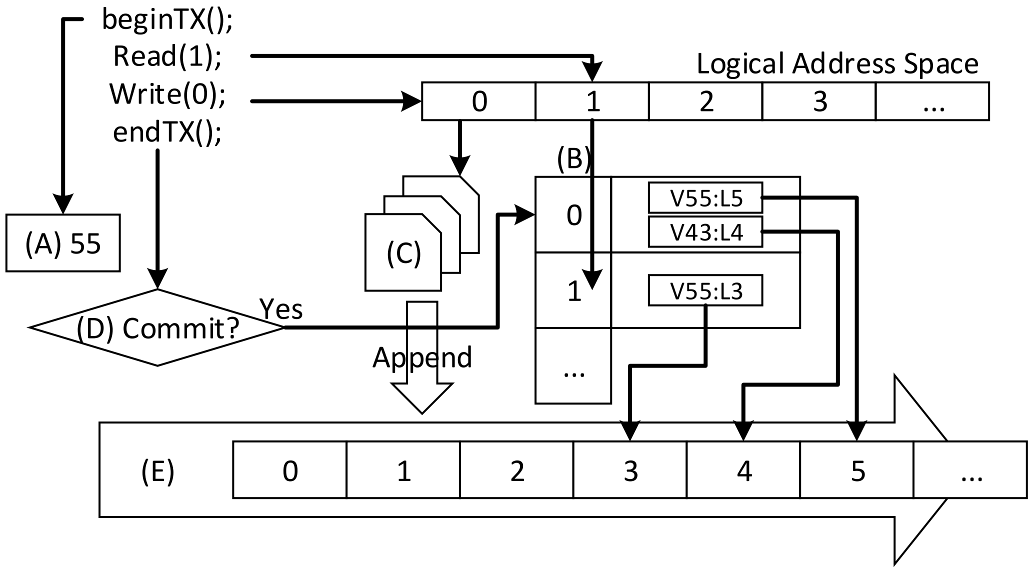


Figure 4.3: Isotope consists of (A) a timestamp counter, (B) a multiversion index, (C) a write buffer, (D) a decision algorithm, and (E) a persistent log.

When the application calls *beginTX*, Isotope creates an in-memory intention record for the speculative transaction: a simple data structure with a start timestamp and a read/write-set. Each entry in the read/write-set consists of a block address, a bitmap that tracks the accessed status of smaller fixed-size chunks or fragments within the block (by default, the fragment size is 16 bytes, resulting in a 256-bit bitmap for each 4KB block), and an additional 4KB payload only in the write-set. These bitmaps are never written persistently and are only maintained in-memory for currently executing transactions. After creating the intention record, the *beginTX* call sets its start timestamp to the current value of the timestamp counter (*A* in Figure 4.3) without incrementing it.

Until *endTX* is called, the transaction executes speculatively against the (potentially stale) snapshot, without any effect on the shared or persistent state of the system. Writes update the write-set and are buffered in-memory (*C* in Figure 4.3) without issuing any I/O. A transaction can read its own buffered writes, but all other reads within the transaction are served from the snapshot corresponding to the start timestamp using the multiversion index (*B* in Figure 4.3). The *mark_accessed* call modifies the bitmap for a previously read or written block to indicate which bits the application actually touched. Multiple *mark_accessed* calls have a cumulative effect on the bitmap. At any point, the transaction can be preemptively aborted by Isotope simply by discarding its intention record and buffered writes. Subsequent reads, writes, and *endTX* calls will be unable to find the record and return an error code to the application.

All actions happen on the *endTX* call, which consist of two distinct phases: *deciding* the commit/abort status of the transaction, and *applying* the transaction (if it commits) to the state of the logical address space. Regardless of how it performs these two phases, the first action taken by *endTX* is to assign the transaction a commit timestamp by reading and incrementing the global counter. The commit timestamp of the transaction is used to make the commit decision, and is also used as the version number for all the writes within the transaction if it commits. We use the terms timestamp and version number interchangeably in the following text.

4.2.1 Deciding Transactions

To determine whether the transaction commits or aborts, *endTX* must detect the existence of conflicting transactions. The isolation guarantee provided – strict serializability or snapshot isolation – depends on what constitutes a conflicting transaction. We first consider a simple strawman scheme that provides strict

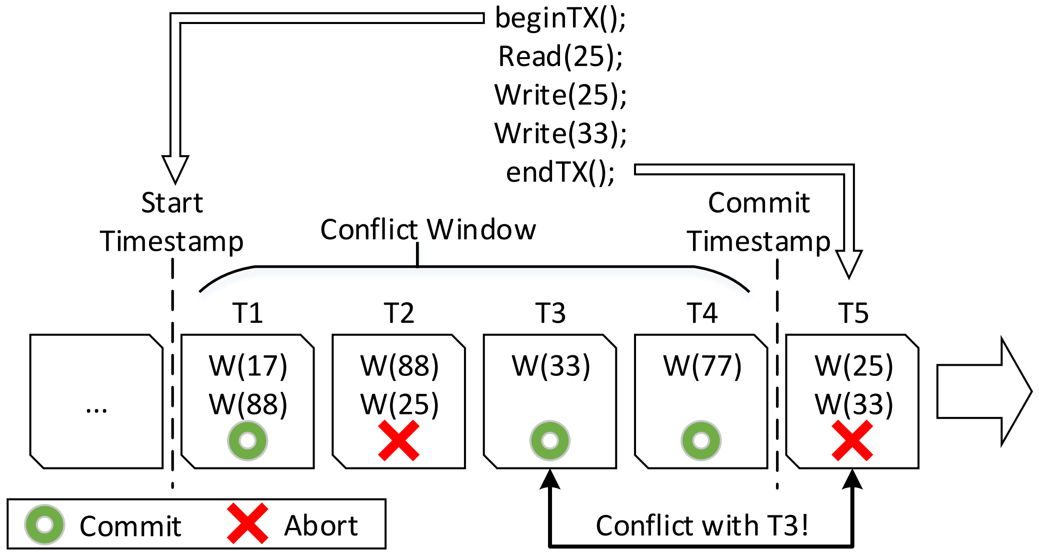


Figure 4.4: Conflict detection under snapshot isolation: a transaction commits if no other committed transaction in its conflict window has an overlapping write-set.

serializability and implements conflict detection as a function of the multiversion index. Here, transactions are processed in commit timestamp order, and for each transaction the multiversion index is consulted to check if any of the logical blocks in its read-set has a version number greater than the current transaction's start timestamp. In other words, we check whether any of the blocks read by the transaction has been updated since it was read.

This scheme is simple, but suffers from a major drawback: the granularity of the multiversion index has to match the granularity of conflict detection. For example, if we want to check for conflicts at 16-byte grain, the index has to track version numbers at 16-byte grain as well; this blows up the size of the in-memory index by 256X compared to a conventional block-granular index. As a result, this scheme is not well-suited for fine-grained conflict detection.

To perform fine-grained conflict detection while avoiding this blow-up in the size of the index, Isotope instead implements conflict detection as a function over the temporal stream of prior transactions (see Figure 4.4). Concretely, each transaction has a conflict window of prior transactions between its start timestamp and its commit timestamp.

- For strict serializability, the transaction T aborts if any committed transaction in its conflict window modified an address that T *read*; else, T commits.
- For snapshot isolation, the transaction T aborts if any committed transaction in its conflict window modified an address that T *wrote*; else, T commits.

In either case, the commit/abort status of a transaction is a function of a window of transactions immediately preceding it in commit timestamp order.

When *endTX* is called on T, a pointer to its intention record is inserted into the slot corresponding to its commit timestamp in an in-memory array. Since the counter assigns contiguous timestamps, this array has no holes; each slot is eventually occupied by a transaction. At this point, we do not yet know the commit/abort status of T and have not issued any write I/O, but we have a start timestamp and a commit timestamp for it. Each slot is guarded by its own lock.

To decide if T commits or aborts, we simply look at its conflict window of transactions in the in-memory array (i.e., the transactions between its start and commit timestamps). We can decide T's status once all these transactions have decided. T commits if each transaction in the window either aborts or has no overlap between its read/write-set and T's read/write-set (depending on the transactional semantics). Since each read/write-set stores fine-grained information about which fragments of the block are accessed, this scheme provides fine-grained conflict detection without increasing the size of the multiversion index.

Defining the commit/abort decision for a transaction as a function of other transactions is a strategy as old as optimistic concurrency control itself [74], but choosing an appropriate implementation is non-trivial. Like us, Bernstein et al. [103] formulate the commit/abort decision for distributed transactions in the Hyder system as a function of a conflict window over a totally ordered stream of transaction intentions. Unlike us, they explicitly make a choice to use the spatial state of the system (i.e., the index) to decide transactions. A number of factors drive our choice in the opposite direction: we need to support writes at arbitrary granularity (e.g., an inode) without increasing index size; our intention log is a local in-memory array and not distributed or shared across the network, drastically reducing the size of the conflict window; and checking for conflicts using read/write-sets is easy since our index is a simple address space.

4.2.2 Applying Transactions

If the outcome of the decision phase is commit, *endTX* proceeds to apply the transaction to the logical address space. The first step in this process is to append the writes within the transaction to the persistent log. This step can be executed in parallel for multiple transactions, as soon as each one's decision is known, since the existence and order of writes on the log signifies nothing: the multiversion index still points to older entries in the log. The second step involves changing the multiversion index to point to the new entries. Once the index has been changed, the transaction can be acknowledged and its effects are visible.

One complication is that this protocol introduces a lost update anomaly. Consider a transaction that reads a block (say an allocation bitmap in a filesystem), examines and changes the first bit, and writes it back. A second transaction reads the same block concurrently, examines and changes the last bit, and writes it back. Our conflict detection scheme will correctly allow both transactions to commit. However, each transaction will write its own version of the 4KB bitmap, omitting the other's modification; as a result, the transaction with the higher timestamp will destroy the earlier transaction's modification. To avoid such lost updates, the *endTX* call performs an additional step for each transaction before appending its buffered writes to the log. Once it knows that the current transaction can commit, it scans the conflict window and *merges* updates made by prior committed transactions to the blocks in its write-set.

4.2.3 Implementation Details

Isotope is implemented as an in-kernel software module in Linux 2.6.38; specifically, as a device mapper that exposes multiple physical block devices as a single virtual disk, at the same level of the stack as software RAID. Below, we discuss the details of this implementation.

Log implementation: Isotope implements the log (i.e., E in Figure 4.3) over a conventional address space with a counter marking the tail (and additional bookkeeping information for garbage collection, which we discuss shortly).

From a correctness and functionality standpoint, Isotope is agnostic to how this address space is realized. For good performance, it requires an implementation that works well for a logging workload where writes are concentrated at the tail, while reads and garbage collection can occur at random locations in the body. A naive solution is to use a single physical disk (or a RAID-0 or RAID-10 array of disks), but garbage collection activity can hurt performance significantly by randomizing the disk arm. Replacing the disks with SSDs increases the cost-to-capacity ratio of the array without entirely eliminating the performance problem: SSDs typically run slower for random workloads than sequential ones (by at least 2X), and perform poorly when exposed to mixed workloads rather than read-only or write-only workloads [123].

As a result, we use a design where a log is chained across multiple disks or SSDs (similar to Gecko in Chapter 3). Chaining the log across drives ensures that garbage collection – which occurs in the body of the log/chain – is separated from the first-class writes arriving at the tail drive of the log/chain. In addition, a commodity SSD is used as a read cache with an affinity for the tail drive of the chain, preventing application reads from disrupting write sequentiality at the tail drive. In essence, this design 'collars' the throughput of the log, pegging write throughput to the speed of a single drive, but simultaneously eliminating the throughput troughs caused by concurrent garbage collection and read activity.

Garbage collection (GC): Compared to conventional log-structured stores, GC is slightly complicated in Isotope by the need to maintain older versions of blocks. Isotope tracks the oldest start timestamp across all ongoing transactions and makes a best-effort attempt to not garbage collect versions newer than this timestamp. In the worst case, any non-current versions can be discarded without compromising safety, by first preemptively aborting any transactions reading from them. The application can simply retry its transactions, obtaining a new, current snapshot. This behavior is particularly useful for dealing with the effects of rogue transactions that are never terminated by the application. The alternative, which we did not implement, is to set a flag that preserves a running transaction's snapshot by blocking new writes if the log runs out of space; this may be required if it's more important for a long-running transaction to finish (e.g., if it's a critical backup) than for the system to be online for writes.

Caching: The *please_cache* call in Isotope allows the application to mark the blocks it wants cached in RAM. To implement caching, Isotope annotates the multiversion index with pointers to cached copies of block versions. This call is merely a hint and provides no guarantees to the application. In practice, our implementation uses a simple LRU scheme to cache a subset of the blocks if the application requests caching indiscriminately.

Index persistence: Thus far, we have described the multiversion index as an in-memory data structure pointing to entries on the log. Changes to the index have to be made persistent so that the state of the system can be reconstructed on failures. To obtain persistence and failure atomicity for these changes, we use a *metadata log*. The size of this log can be limited via periodic checkpoints.

A simple option is to store the metadata log on battery-backed RAM, or on newer technologies such as PCM or flash-backed RAM (e.g., Fusion-io's Auto-Commit Memory [18]). In the absence of special hardware on our experimental testbed, we instead used a commodity SSD. Each transaction's description in the metadata log is quite compact (i.e., the logical block address and the physical log position of each write in it, and its commit timestamp). To avoid the slowdown and flash wear-out induced by logging each transaction separately as a synchronous page write, we batch multiple committed transactions together [52], delaying the final step of modifying the multiversion index and acknowledging the transaction to the application. We do not turn off the write cache on the SSD, relying on its ability to flush data on power failures using supercapacitors.

Memory overhead: A primary source of memory overhead in Isotope is the multiversion index. A single-version index that maps a 2TB logical address space to an 4TB physical address space can be implemented as a simple array that requires 2GB of RAM (i.e., half a billion 4-byte entries), which can be easily maintained in RAM on modern machines. Associating each address with a version (without supporting access to prior versions) doubles the space requirement to 4GB (assuming 4-byte timestamps), which is still feasible. However, multiversioned indices that allow access to past versions are more expensive, due to the fact that multiple versions need to be stored, and because a more complex data structure is required instead of an array with fixed-size values. These concerns are mitigated by the fact that Isotope is not designed to be a fully-fledged multiversion store; it only stores versions from the recent past, corresponding to the snapshots seen by executing transactions.

Accordingly, Isotope maintains a pair of indices: a single-version index in the form of a simple array and a multiversion index implemented as a hashtable. Each entry in the single-version index either contains a valid physical address if the block has only one valid, non-GC'ed version, a null value if the block has never been written, or a constant indicating the existence of multiple versions. If a transaction issues a read and encounters this constant, the multiversion index is consulted. An address is moved from the single-version index to the multi-version index when it goes from having one version to two; it is moved back to the single-version index when its older version(s) are garbage collected (as described earlier in this section).

The multiversion index consists of a hashtable that maps each logical address to a linked list of its existing versions, in timestamp order. Each entry contains forward and backward pointers, the logical address, the physical address, and the timestamp. A transaction walks this linked list to find the entry with the highest timestamp less than its snapshot timestamp. In addition, the entry also has a pointer to the in-memory cached copy, as described earlier. If an address is cached, the first single-version index is marked as having multiple versions even if it does not, forcing the transaction to look at the hashtable index and encounter the cached copy. In the future, we plan on applying recent work on compact, concurrent maps [54] to further reduce overhead.

Rogue Transactions: Another source of memory overhead in Isotope is the buffering of writes issued by in-progress transactions. Each write adds an entry to the write-set of the transaction containing the 4KB payload and a $\frac{4K}{C}$ bit bitmap, where C is the granularity of conflict detection (e.g., with 16-byte detection, the bitmap is 256 bits). Rogue transactions that issue a large number of writes are a concern, especially since transactions can be exposed to end-user applications. To handle this, Isotope provides a configuration parameter to set the maximum number of writes that can be issued by a transaction (set to 256 by default); beyond this, writes return an error code. Another parameter sets the maximum number of outstanding transactions a single process can have in-flight (also set to 256). Accordingly, the maximum memory a rogue process can use within Isotope for buffered writes is roughly 256MB. When a process is killed, its outstanding transactions are preemptively aborted.

Despite these safeguards, it is still possible for Isotope to run out of memory if many processes are launched concurrently and each spams the system with rogue, never-ending transactions. In the worst case, Isotope can always relieve memory pressure by preemptively aborting transactions. Another option which we considered is to flush writes to disk before they are committed; since the metadata index does not point to them, they won't be visible to other transactions. Given that the system is only expected to run out of memory in pathological cases where issuing I/O might worsen the situation, we didn't implement this scheme.

Note that the in-memory array that Isotope uses for conflict detection is not a major source of memory overhead; pointers to transaction intention records are inserted into this array in timestamp order only after the application calls *endTX*, at which point it has relinquished control to Isotope and cannot prolong the transaction. As a result, the lifetime of an entry in this array is short and limited to the duration of the *endTX* call.

Clustering sub-block writes: Widening the interface to the block store can enable new optimizations. For example, applications often issue small, sub-block writes to their on-disk metadata structures (e.g., a filesystem might modify an inode within a block). In our current implementation, as in conventional block stores, each sub-block write triggers a full block write to the data log.

Instead, Isotope can leverage its knowledge of the dirty regions within each write to optimize disk I/O in the critical path of the transaction. When applying committed transactions, it can accumulate sub-block writes from different logical blocks into a single physical block on the persistent data log; in effect, this physical block acts as a byte-level mini-log, temporally clustering small writes. For example, a filesystem write might involve a single bit flip on an allocation map, a pointer assignment in an indirection block and a timestamp change on an inode; all three of these fine-grained modifications could be written out in a single 4KB mini-log write to the persistent log. Eventually, the writes have to be rewritten as separate, conventional entries on the log. In the interim, the multiversion index has to track which fragments exist in these mini-log entries to serve reads correctly.

4.3 Isotope Applications

To illustrate the usability and performance of Isotope, we built four applications using Isotope transactions: IsoHT, a key-value store built over a persistent hashtable; IsoBT, a key-value store built over a persistent B-tree; IsoFS, a user-space POSIX filesystem; and ImgStore, an image storage service that stores images in IsoHT, and a secondary index in IsoBT. These applications implement each call in their respective public APIs by following a simple template that wraps the entire function in a single transaction, with a retry loop in case the transaction aborts due to a conflict (see Figure 4.2).

4.3.1 Transactional Key-Value Stores

Library-based or 'embedded' key-value stores (such as LevelDB or Berkeley DB) are typically built over persistent, on-disk data structures. We built two key-value stores called IsoHT and IsoBT, implemented over an on-disk hashtable and B-tree data structure, respectively. Both key-value stores support basic put/get operations on key-value pairs, while IsoBT additionally supports range queries. Each API call is implemented via a single transaction of block reads and writes to an Isotope volume.

We implemented IsoHT and IsoBT in three stages. First, we wrote code without Isotope transactions, using a global lock to guard the entire hashtable or B-tree. The resulting key-value stores are functional but slow, since all accesses are serialized by the single lock. Further, they do not provide failure atomicity: a crash in the middle of an operation can catastrophically violate data structure integrity.

In the second stage, we simply replaced the acquisitions/releases on the global lock with Isotope *beginTX*/*endTX*/ *abortTX* calls, without changing the overall number of lines of code. With this change, the key-value stores provide both fine-grained concurrency control (at block granularity) and failure atomicity. Finally, we added optional *mark_accessed* calls to obtain sub-block concurrency control, and *please_cache* calls to cache the data structures (e.g., the nodes of the B-tree, but not the values pointed to by them). Table 4.1 reports on the lines of code (LOC) counts at each stage for the two key-value stores.

Application	Original with locks	Basic APIs (lines modified)	Optional APIs (lines added)
IsoHT	591	591 (15)	617 (26)
IsoBT	1,229	1,229 (12)	1,246 (17)
IsoFS	997	997 (19)	1,022 (25)

Table 4.1: Lines of code for Isotope storage systems.

4.3.2 Transactional Filesystem

IsoFS is a simple user-level filesystem built over Isotope accessible via FUSE [6], comprising 1K lines of C code. Its on-disk layout consists of distinct regions for storing inodes, data, and an allocation bitmap for each. Each inode has an indirect pointer and a double indirect pointer, both of which point to pages allocated from the data region. Each filesystem call (e.g., *setattr*, *lookup*, or *unlink*) uses a single transaction to access and modify multiple blocks. The only functionality implemented by IsoFS is the mapping and allocation of files and directories to blocks; atomicity, isolation, and durability are handled by Isotope.

IsoFS is stateless, caching neither data nor metadata across filesystem calls (i.e., across different transactions). Instead, IsoFS tells Isotope which blocks to cache in RAM. This idiom turned out to be surprisingly easy to use in the context of a filesystem; we ask Isotope to cache all bitmap blocks on startup, each inode block when an inode within it is allocated, and each data block that's allocated as an indirect or double indirect block. Like the key-value stores, IsoFS was implemented in three stages and required few extra lines of code to go from a global lock to using the Isotope API (see Table 4.1).

IsoFS trivially exposes transactions to end applications over files and directories. For example, a user might create a directory, move a file into it, edit the file, and rename the directory, only to abort the entire transactions and revert the filesystem to its earlier state. One implementation-related caveat is that we were unable to expose transactions to end applications of IsoFS via the FUSE interface, since FUSE decouples application threading from filesystem threading and does not provide any facility for explicitly transferring a transaction handle on each call. Accordingly, we can only expose transactions to the end application if IsoFS is used directly as a library within the application's process.

4.3.3 Experience

Composability: As we stated earlier, Isotope-based storage systems are trivially composable: a single transaction can encapsulate calls to IsoFS, IsoHT and IsoBT. To demonstrate the power of such composability, we built ImgStore, the image storage application described in Section 4.1. ImgStore stores images in IsoHT, using 64-bit IDs as keys. It then stores a secondary index in IsoBT, mapping dates to IDs. The implementation of ImgStore is trivially simple: to add an image, it creates a transaction to put the image in IsoHT, and then updates the secondary index in IsoBT. The result is a storage system that – in just 148 LOC – provides hashtable-like performance for gets while supporting range queries.

Isolation Levels: Isotope provides both strict serializability and snapshot isolation; our expectation was that developers would find it difficult to deal with the semantics of the latter. However, our experience with IsoFS, IsoHT and IsoBT showed otherwise. Snapshot isolation provides better performance than strict serializability, but introduces the write skew anomaly [36]: if two concurrent transactions read two blocks and each updates one of the blocks (but not the same one), they will both commit despite not being serializable in any order. The write skew anomaly is problematic for applications if a transaction is expected to maintain an integrity constraint that includes some block it does not write to (e.g., if the two blocks in the example have to sum to less than some constant). In the case of the storage systems we built, we did not encounter these kinds of constraints; for instance, no particular constraint holds between different bits on an allocation map. As a result, we found it relatively easy to reason about and rule out the write skew anomaly.

Randomization: Our initial implementations exhibited a high abort rate due to deterministic behavior across different transactions. For example, a simple algorithm for allocating a free page involved getting the first free bit from the allocation bitmap; as a result, multiple concurrent transactions interfered with each other by trying to allocate the same page. To reduce the abort rate, it was sufficient to remove the determinism in simple ways; for example, we assigned each thread a random start offset into the allocation bitmap.

4.4 Performance Evaluation

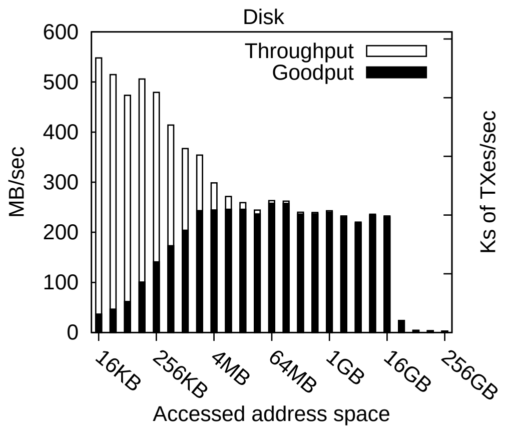
We evaluate Isotope on a machine with an Intel Xeon CPU with 24 hyper-threaded cores, 24GB RAM, three 10K RPM disks of 600GB each, an 128GB SSD for the OS and two other 240GB SSDs with SATA interfaces. In the following experiments, we used two primary configurations for Isotope's persistent log: a three-disk chained logging instance with a 32GB SSD read cache in front, and a 2-SSD chained logging instance. In some of the experiments, we compare against conventional systems running over RAID-0 configurations of 3 disks and 2 SSDs, respectively. In the chained logging configurations, all writes are logged to the single tail drive, while reads are mostly served by the other drives (and the SSD read cache for the disk-based configuration). The performance of this logging design under various workloads and during GC activity has been documented in Chapter 3. In all our experiments, GC is running in the background and issuing I/Os to the drives in the body of the chain to compact segments, without disrupting the tail drive.

Our evaluation consists of two parts. First, we focus on the performance and overhead of Isotope, showing that it exploits fine-grained concurrency in workloads and provides high, stable throughput. Second, we show that Isotope applications – in addition to being simple and robust – are fast, efficient, and composable into larger applications.

4.4.1 Isotope Performance

To understand how Isotope performs depending on the concurrency present in the workload, we implemented a synthetic benchmark. The benchmark executes a simple type of transaction that reads three randomly chosen blocks, modifies a random 16-byte segment within each block (aligned on a 16-byte boundary), and writes them back. This benchmark performs identically with strict serializability and snapshot isolation, since the read-set exactly matches the write-set.

In the following experiments, we executed 64 instances of the micro benchmark concurrently, varying the size of the address space accessed by the instances to vary contention. The blocks are chosen from a specific prefix of the



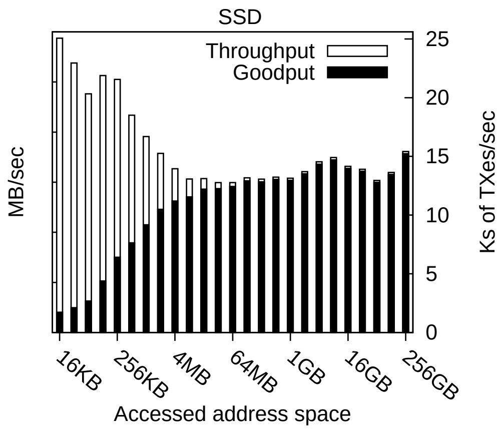


Figure 4.5: Without fine-grained conflict detection, Isotope performs well under low contention workloads.

address space, which is a parameter to the benchmark; the longer this prefix, the bigger the fraction of the address space accessed by the benchmark, and the less skewed the workload. The two key metrics of interest are transaction goodput (measured as the number of successfully committed transactions per second, as well as the total number of bytes read or written per second by these transactions) and overall transaction throughput; their ratio is the commit rate of the system. Each data point in the following graphs is averaged across three runs; in all cases, the minimum and the maximum run were within 10% of the average.

Figure 4.5 shows the performance of this benchmark against Isotope without fine-grained conflict detection; i.e., the benchmark does not issue *mark_accessed* calls for the 16-byte segments it modifies. On the x-axis, we increase the fraction of the address space accessed by the benchmark. On the left y axis, we plot the rate at which data is read and written by transactions; on the right y-axis, we plot the number of transactions/sec. On both disk and SSD, transactional contention cripples performance on the left part of the graph: even though the

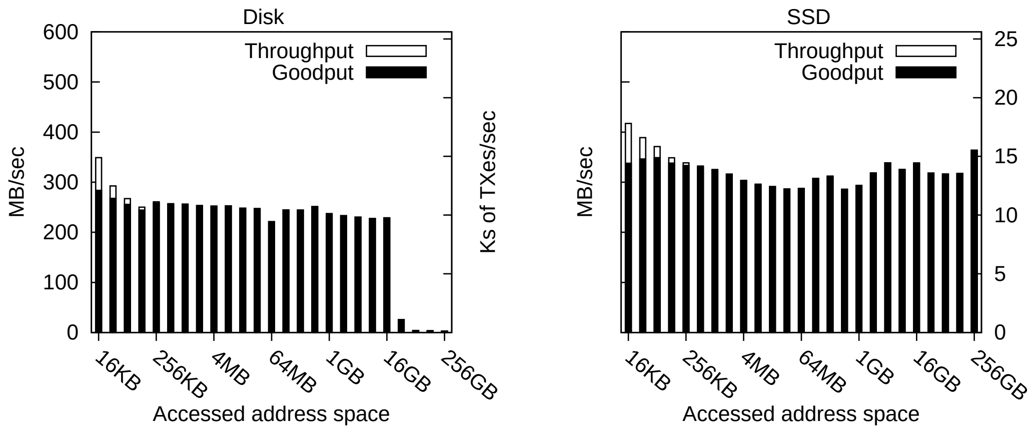


Figure 4.6: With fine-grained conflict detection, Isotope performs well even under high block-level contention.

benchmark attempts to commit thousands of transactions/sec, all of them access a small number of blocks, leading to low goodput. Note that overall transaction throughput is very high when the commit rate is low: aborts are cheap and do not result in storage I/O.

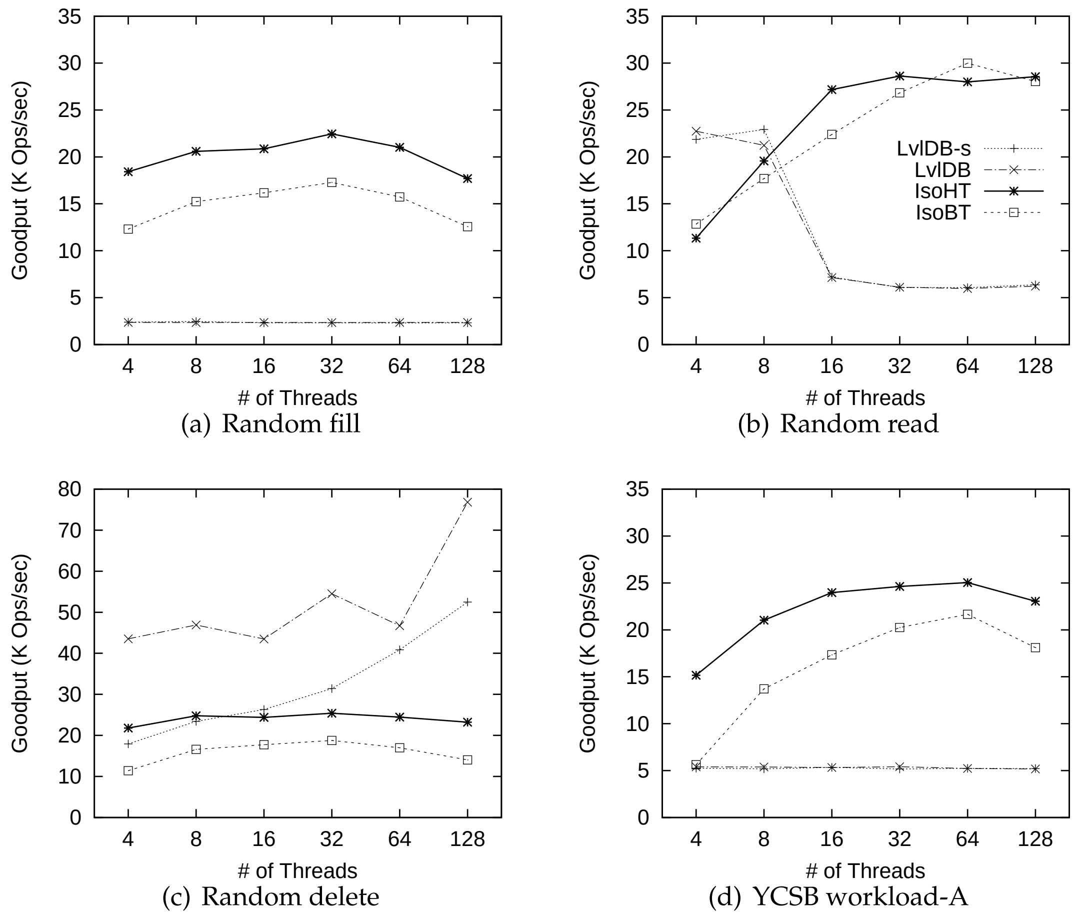
Conversely, disk contention hurts performance on the right side of Figure 4.5-Left: since the blocks read by each transaction are distributed widely across the address space, the 32GB SSD read cache is ineffective in serving reads and the disk arm is randomized and seeking constantly. As a result, the system provides very few transactions per second (though with a high commit rate). In the middle of the graph is a sweet spot where Isotope saturates the disk at roughly 120 MB/s of writes, where the blocks accessed are concentrated enough for reads to be cacheable in the SSD (which supplies 120 MB/s of reads, or 30K 4KB IOPS), while distributed enough for writes to not trigger frequent conflicts.

We can improve performance on the left side of the graphs in Figure 4.5 via fine-grained conflict detection. In Figure 4.6, the benchmark issues *mark_accessed* calls to tell Isotope which 16-byte fragment it is modifying. The result is high, stable goodput even when all transactions are accessing a small number of blocks, since there is enough fragment-level concurrency in the system to ensure a high commit rate. Using the same experiment but with smaller and larger data access and conflict detection granularities than 16 bytes showed similar trends. Isotope's conflict detection was not CPU-intensive: we observed an average CPU utilization of 5.96% without fine-grained conflict detection, and 6.17% with it.

4.4.2 Isotope Application Performance

As described earlier, we implemented two key-value stores over Isotope: IsoHT using a hashtable index and IsoBT using a B-tree index, respectively. IsoBT exposes a fully functional LevelDB API to end applications; IsoHT does the same minus range queries. To evaluate these systems, we used the LevelDB benchmark [11] as well as the YCSB [46] benchmark. We ran the fill-random, read-random, and delete-random workloads of the LevelDB benchmark and YCSB workload-A traces (50% reads and 50% updates following a zipf distribution on keys). All these experiments are on the 2-SSD configuration of Isotope. For comparison, we ran LevelDB on a RAID-0 array of the two SSDs, in both synchronous mode ('LvlDB-s') and asynchronous mode ('LvlDB'). LevelDB was set to use no compression and the default write cache size of 8MB. For all the workloads, we used a value size of 8KB and varied the number of threads issuing requests from 4 to 128. Results with different value sizes (from 4KB to 32KB) showed similar trends.

For operations involving writes (Figure 4.7-(a), (c), and (d)), IsoHT and IsoBT



(a) Random fill (b) Random read

(c) Random delete (d) YCSB workload-A

Figure 4.7: IsoHT and IsoBT outperform LevelDB for data operations while providing stronger consistency.

goodput increases with the number of threads, but dips slightly beyond 64 threads due to an increased transaction conflict rate. For the read workload (Figure 4.7-(b)), throughput increases until the underlying SSDs are saturated. Overall, IsoHT has higher goodput than IsoBT, since it touches fewer metadata blocks per operation. We ran these experiments with Isotope providing snapshot isolation, since it performed better for certain workloads and gave sufficiently strong semantics for building the key-value stores. With strict serializability, for instance, the fill workload showed nearly identical performance, whereas the delete workload ran up to 25% slower.

LevelDB's performance is low for fill operations due to sorting and multi-level merging (Figure 4.7-(a)), and its read performance degrades as the number of concurrent threads increases because of the CPU contention in the skip list, cache thrashing, and internal merging operations (Figure 4.7-(b)). Still, LevelDB's delete is very efficient because it only involves appending a small delete intention record to a log, whereas IsoBT/IsoHT has to update a full 4KB block per delete (Figure 4.7-(c)).

The point of this experiment is not to show IsoHT/IsoBT is better than LevelDB, which has a different internal design and is optimized for specific workloads such as sequential reads and bulk writes. Rather, it shows that systems built over Isotope with little effort can provide equivalent or better performance than an existing system that implements its own concurrency control and failure atomicity logic.

Composability

To evaluate the composability of Isotope-based storage systems, we ran the YCSB workload on ImgStore, our image storage application built over IsoHT and IsoBT. In our experiments, ImgStore transactionally stored a 16KB payload (corresponding to an image) in IsoHT and a small date-to-ID mapping in IsoBT. To capture the various ways in which Isotope storage systems can be composed (see Section 4.1), we implemented several versions of ImgStore: cross-library, where ImgStore accesses the two key-value stores as in-process libraries, with each transaction executing within a single user-space thread; cross-thread, where ImgStore accesses each key-value store using a separate thread, and requires transactions to execute across them; and cross-process, where each

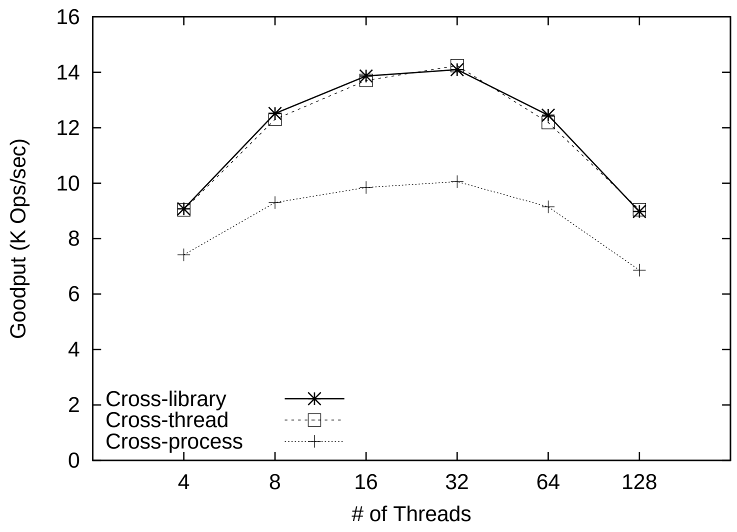


Figure 4.8: YCSB over different compositions of IsoBT and IsoHT.

key-value store executes within its own process and is accessed by ImgStore via socket-based IPC. Figure 4.8 shows the resulting performance for all three versions. It shows that the cost of the extra *takeoverTX*/*releaseTX* calls required for cross-thread transactions is negligible. As one might expect, cross-process transactions are slower due to the extra IPC overhead. Additionally, ImgStore exhibits less concurrency than IsoHT or IsoBT (peaking at 32 threads), since each composite transaction conflicts if either of its constituent transactions conflict.

Filesystem Performance

Next, we compare the end-to-end performance of IsoFS running over Isotope using the IOZone [9] write/rewrite benchmark with 8 threads. Each thread writes to its own file using a 16KB record size until the file size reaches 256MB; it then rewrites the entire file sequentially; and then rewrites it randomly. We ran this workload against IsoFS running over Isotope, which converted each 16KB write into a transaction involving four 4KB Isotope writes, along with

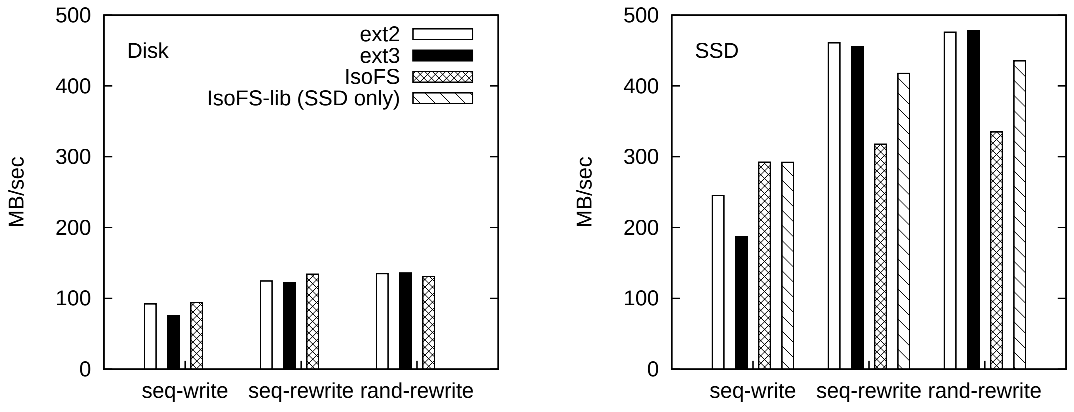


Figure 4.9: IOZone over IsoFS and ext2/ext3.

metadata writes. We also ran ext2 and ext3 over Isotope; these issued solitary, non-transactional reads and writes, which were interpreted by Isotope as singleton transactions (in effect, Isotope operated as a conventional log-structured block store, so that ext2 and ext3 are not penalized for random I/Os). We ran ext3 in 'ordered' mode, where metadata is journaled but file contents are not.

Figure 4.9 plots the throughput observed by IOZone: on disk, IsoFS matches or slightly outperforms ext2 and ext3, saturating the tail disk on the chain. On SSD, IsoFS is faster than ext2 and ext3 for initial writes, but is bottlenecked by FUSE on rewrites. When we ran IsoFS directly using a user-space benchmark that mimics IOZone ('IsoFS-lib'), throughput improved to over 415MB/s. A secondary point made by this graph is that Isotope does not slow down applications that do not use its transactional features (the high performance is mainly due to the underlying logging scheme, but ext2 and ext3 still saturate disk and SSD for rewrites), satisfying a key condition for pushing functionality down the stack [108].

4.5 Summary

In this chapter, we explored a block-level transaction support abstraction to enable transactional isolation in cloud storage servers. We described Isotope as an instance of this approach to achieve the transactional abstraction. Isotope is a transactional block store that provides isolation in addition to atomicity and durability. We showed that transactional isolation can be implemented efficiently within the block layer, leveraging the inherent multi-versioning of log-structured block stores and application-provided hints for fine-grained conflict detection. Isotope-based systems are simple and fast, while obtaining database-strength guarantees on failure atomicity, durability, and consistency. Isotope-based systems are also composable, allowing application-initiated transactions to span multiple storage systems and different abstractions such as files and key-value pairs. The portability of Isotope allows easy design of transactional applications in any layer of the storage stack and it can be easily used by any application running in a cloud storage server.

CHAPTER 5

CONSISTENCY CONTROL IN CLOUD SERVERS USING STALE DATA

The control of consistency within cloud storage servers is necessary as the server becomes more powerful and complex. The ongoing explosion in the diversity of memory and storage technology has made hardware heterogeneity a fact of life for cloud storage servers. Current storage system designs typically use a mix of multi-device idioms – such as caching, tiering, striping, mirroring, etc. – to spread data across a range of devices, including hard disks, DRAM, NAND-based solid state drives (SSDs), and byte-addressable NVRAM. Each such storage medium exhibits vastly different throughput and latency characteristics; access latencies to data can vary considerably depending on which media the data resides on. Figure 5.1 shows the performance characteristics and cost of some of the storage options available at the time of writing this dissertation.

In parallel, multi-device storage systems have become increasingly multi-versioned, retaining older versions of data that are typically not exposed to the application. Often, multi-versioning is a side-effect of log-structured designs that avoid writing in place; for example, SSDs expose a single-version block ad-

Devices	Throughput	Latency	Cost / GB
Registers	-	1 cycle	-
Caches	-	2-10ns	-
DRAM	10s of GB/s	100-200ns	$10.00
NVDIMM	10s of GB/s	100-200ns	$10.00
NVMM	10s of GB/s	800ns	$5.00
NVMe	2GB/s	10-100us	$1.40
SATA SSD	500MB/s	400us	$0.40
Disk	100MB/s	10ms	$0.05

Figure 5.1: The new storage/memory hierarchy (from a LADIS 2015 talk by Andy Warfield).

dress space to applications, but internally log data to avoid triggering expensive erase operations on block rewrites. In other cases, tiering or caching strategies can introduce multiple versions by replicating data and synchronizing lazily; for example, SSDs typically have DRAM-based write caches that are lazily flushed to the underlying flash.

We observe that the existence of multiple versions of data within a storage system – and the non-uniform performance characteristics of the storage media that these versions reside on – creates an opportunity for trading off consistency or staleness for performance. We make a case for *weakly consistent local storage systems* in a cloud storage server: when applications access data, we want the option of providing them with stale data in exchange for better performance. This behavior is in contrast to the strong consistency or linearizability offered by existing storage systems, which guarantee that read operations will reflect all writes that complete before the read was issued [68]. Accessing older versions can provide better performance for a number of reasons: the latest version might be slow to access because it resides on a write cache that is unoptimized for reads [124], or on a hard disk stripe that is currently logging writes [119] or undergoing maintenance operations such as a RAID rebuild. In all these cases, accessing older versions can provide superior latency and/or throughput. In Section 2.1.3, we have described these and other scenarios in detail.

The killer app for a weakly consistent local storage system is distributed cloud storage. Services such as S3, DynamoDB, and Windows Azure Storage routinely negotiate weak consistency guarantees with clients, primarily to mask round-trip delays to remote data centers. A client might request client-centric consistency, such as read-my-writes consistency, monotonic reads, or bounded-writes consistency from the cloud service, indicating its willingness to tolerate an older version of data for better performance. For example, in the case of monotonic reads consistency, a client that last saw version 100 of a key is satisfied with any version of that key equal to or greater than 100.

Traditionally, a distributed storage service leverages weaker consistency requirements to direct the client's request to nearby servers that can provide the desired consistency (e.g., contain a version equal to or greater than 100). The server itself – typically implemented as a user-level process over a strongly consistent local storage subsystem – strenuously returns the latest value of the key that it stores (e.g., version 200), ignoring the presence of older, potentially faster versions on the underlying subsystem that would satisfy the guarantee (e.g., version 110, 125, etc.). Instead, a cloud storage service could propagate knowledge of weaker consistency requirements down to the local storage subsystem on each individual server, allowing it to return older data at faster speeds.

Accordingly, in this chapter we propose and explore a new class of local storage systems for cloud storage servers – e.g. embedded key-value stores, filesystems, and block stores – that are consistency-aware, trading off staleness for performance. We call these StaleStores. While different StaleStores can have widely differing external APIs and internal designs, they share a number of common features, such as support for multi-versioned access, and cost estimation APIs that allow applications to determine the fastest version for a particular data item.

In addition, we describe the design and implementation of a particular StaleStore: a log-structured block store called Yogurt. We implement over Yogurt a variant of a distributed cloud storage system called Pileus [131] that supports multiple consistency levels, and show that exploiting the performance/consistency trade-off within individual cloud storage servers provides a 6X speed-up in access latency.

5.1 Design Space for StaleStores

For any cloud storage service, the software stack on a single server typically contains a top-most layer that runs as a user-space process and exposes some high-level abstraction – such as key-value pairs and files – over the network to applications running on remote clients. This process acts in concert with other processes to implement a distributed storage service; for example, it might act as a primary or secondary replica, or as a caching node. If the distributed storage service supports weaker consistency guarantees, clients can mandate that reads satisfy some such guarantee (such as read-your-writes or monotonic reads); typically they do so by specifying a set of versions which are permissible. Many systems rely on *timestamps* that provide an ordering across versions; reads can then specify the earliest timestamp they can tolerate without violating the required guarantee.

As a concrete example of a cloud storage service that supports weaker consistency levels, the Pileus system [131] consists of a single primary server and multiple backup servers. A client writes a new key-value pair by sending it to the primary, which assigns a monotonically increasing timestamp to it before writing it to local storage. The primary then asynchronously sends the update to the backups, which apply updates in timestamp order. As a result, a global ordering exists across all updates (and consequently all versions of data). At any given point in time, each backup server contains a strict prefix of this global order corresponding to some timestamps. Clients can then obtain weaker consistency guarantees by specifying a timestamp for their reads, and contacting the closest server that is storing a prefix which extends beyond this timestamp.

For example, a client has written a key-value pair at the primary and was told by the primary that the write's timestamp is T_{44}. The primary has seen 100 writes, including the client's write, and assigned them timestamps T_1 to T_{100}. Backup A has seen writes up to timestamp T_{50}. Backup B has seen writes up to T_{35}. The client then wishes to issue a read on the same key K satisfying the read-your-writes guarantee; i.e., it requires the read to reflect its own latest write, but not necessarily the latest writes by other clients. Accordingly, it contacts the backup server closest to it with a read request annotated with T_{44}. Backup B cannot satisfy this request since it has seen writes only up to T_{35}. Backup A, on the other hand, can satisfy this request by returning any version of K with a timestamp higher than or equal to T_{44}.

In current distributed storage services, each individual server is typically single-versioned (unless the distributed service exposes reads to older versions as a feature). Specifically, existing systems do not have individual servers selectively returning older versions in order to gain better performance from their local storage stack. In the example above, we want backup A to be capable of selecting a version between T_{44} and T_{50} that can be returned the fastest from its local storage. This is the capability we seek to explore.

Key-Value StaleStore API	**Parameters**	**Description**
Get	key, version #	Reads a key corresponding to the version #.
Put	key, version #, value	Writes a key with the specified value and version #.
GetCost	key, version #	Returns an integer cost to access the specified key with the version #.
GetVersionRange	key, version #	Returns a range of version #s within which a version of a key is valid.

Table 5.1: Example key-value StaleStore.

5.1.1 What Is a StaleStore?

Abstractly, a StaleStore is a single-node storage system that maintains and serves multiple versions. Different StaleStores support different application-facing APIs – such as files, key-value pairs, block storage, etc. – that are augmented in similar ways to allow applications to trade off consistency for performance.

In designing the StaleStore abstraction, we observe that the information required to support consistency and performance trade-offs is typically split between the application and the store. The application (i.e., the server process implementing the distributed cloud store) understands consistency (i.e., timestamps), and the store understands performance characteristics (i.e., where data is placed and how fast it can be accessed). Required is an API that allows performance information to flow up the stack and consistency information to flow down the stack. Specifically, we push consistency information down the stack by associating versions within the multi-version store with application-level timestamps; conversely, we push performance information up the stack by allowing applications to query the estimated cost of issuing a read operation against a specific version.

Accordingly, a StaleStore API has four characteristics. In the following descriptions, we use the terms 'timestamp' and 'version number' interchangeably. In addition, we use the term 'snapshot' to define a consistent view of the data store from the viewpoint of the storage at a particular timestamp.

- **Timestamped writes:** First, writes to the StaleStore are accompanied by a monotonically increasing timestamp. This version number is global across all writes to the StaleStore; for example, for a key-value store, each put operation must have a non-decreasing timestamp, regardless of which key-value pair it touches.
- **Snapshot reads:** Second, the application should be able to read from a consistent, potentially stale snapshot corresponding to a timestamp. Read APIs are augmented with a timestamp parameter. A read operation at a timestamp T reflects all writes with a lower or equal timestamp. For example, for a key-value store, if a particular key has been updated by three puts at timestamps T_7, T_{33} and T_{56} respectively, a get operation at timestamp T_{100} will return the value inserted by the put at T_{56}, which reflects the latest update at timestamp T_{100}.
- **Cost estimation:** Third, the application should be able to query the cost of issuing a particular read operation at a snapshot. This cost is an arbitrary integer value that may not correspond to real-world metrics such as latency or throughput; all we require is that two cost estimates from the same StaleStore can be compared.
- **Version exploration:** Finally, the application should be able to determine – having read a particular version of an item – what range of timestamps

that version is valid for. For example, if the application reads an item X at timestamp T_7, and that item does not change next until timestamp T_{33}, the application can optimize cost querying operations with this information, or read other items at any timestamp in between and still obtain a consistent snapshot across items.

Table 5.1 shows an example API for a key-value StaleStore. It provides an API for timestamped writes (*Put*), snapshot reads (*Get*), cost estimation (*GetCost*), and version exploration (*GetVersionRange*).

Why timestamps instead of consistency guarantees? Making the single-node store aware of individual guarantees (such as read-my-writes or monotonic reads consistency) is challenging; these guarantees can be application-specific and refer to application-level entities (e.g., the session consistency guarantee requires a notion of an application-level session started by a specific client). In contrast, timestamps are compact, simple and sufficient representations of consistency requirements, and are used by a wide range of systems to provide weak consistency in a distributed setting. The higher layer simply tags every read and write with a global timestamp.

What about concurrency control? One approach to implement the above API in a real system involves guarding all data with a single, coarse lock. In this case, it's simple for application logic to ensure that writes are always in non-decreasing timestamp order, and that reads reflect writes with prior timestamps. In practice, however, the application can use fine-grained locking to issue requests in parallel, while providing the same semantics as a single lock. For example, in a key-value store, puts to different keys can proceed in parallel, while a get on a key has to be ordered after any puts to that key with a lower timestamp. We expect the application to implement concurrency control above the StaleStore API (in much the same way a filesystem implements locking above a block store API, or a key-value store implements locking above a filesystem API), while ensuring that the semantics of the system are as if a single lock guards all data.

5.1.2 Which Layer Should Be a StaleStore?

The API exposed by an individual server within a cloud storage service to external clients typically mirrors the API of the cloud storage service. For example, a storage service might expose a key-value API to applications allowing them to put and get key-value pairs; each individual server exposes the same API to client-side logic used by the application to access the service. We call this the *public-facing API*.

Internally, each server runs a process (the application from the StaleStore's perspective) that implements the public-facing API over some internal, single-server storage API; we call this the *internal API*. The internal API could be provided by a filesystem like ext3, an embedded key-value store like LevelDB or RocksDB, a single block store such as Storage Spaces. These are the internal APIs that we propose augmenting to support consistency/performance tradeoffs, as described above. Each of these internal subsystems could be a multi-versioned StaleStore, allowing the application to request older versions from them in exchange for better performance. Alternatively, the application could be implemented over one or more unmodified, single-versioned storage subsystems, and itself act as an application-level StaleStore, managing older versions and accessing the fastest one. Below, we discuss the implications of each option:

Application-level StaleStore: In this option, the application-level storage system manages and maintains versions across unmodified single-version stores (filesystems, key-value stores, block devices), with no support from the underlying local storage stack. This approach has one significant benefit: the application is aware of the consistency guarantee required (or equivalently, of high-level constructs such as timestamps), and knows which versions will satisfy the read. It also has a significant drawback: the application is a user-space process that typically has little visibility or control over the location of data on physical devices. Multiple layers of indirection – in the form of logs, read caches and write caches – can exist between the application and raw hardware. While the application can explicitly maintain versions over a logical address space (a file or a block device), it cannot predict access latencies to individual addresses on each address space.

Filesystem / embedded key-value StaleStore: In this option, the application stores all its data in a filesystem or embedded key-value StaleStore. An important benefit of such an approach is generality and reusability: a filesystem StaleStore can be reused by multiple cloud storage systems. On the flip side, it itself operates over a logical address space – a block device – and has little visibility into where blocks are mapped, making it difficult to estimate the cost of reads to particular versions. This is particularly true with the advent of 'smart' block layers in hardware (e.g. SSDs) and software (e.g. Microsoft's Storage Spaces), which are sophisticated, multi-device systems in themselves. However, certain types of StaleStores can only be implemented at the filesystem or key-value store level; one example is scenario S6 from Section 2.1.3, in which

StaleStore APIs	**Parameters**	**Description**
ReadVersion	Block address, version #.	Reads a block corresponding to the version #.
WriteVersion	Block address, version #, data.	Writes a block with the specified version #.
GetCost	Block address, version #.	Returns an integer cost to access the specified block with the version #.
GetVersionRange	Block address, version #.	Returns the snapshot version range where the block data is intact.

Wrapper APIs	**Parameters**	**Description**
POSIX APIs		Does basic block I/Os such as read, write, seek, etc.
OpenSnapshot	Version #	Opens snapshot.
CloseSnapshot		Closes snapshot and flushes writes.

Table 5.2: Yogurt APIs.

a key-value store combines fine-grained logging with a block-grain buffer cache over a block address space with relatively uniform access latencies.

Block-level StaleStore: The third option is for a smart block layer to manage, maintain, and expose versions. The block layer has detailed information on where each block in its address space lives, and can provide accurate access latency estimates. Further, the block device shares the advantage of the filesystem: implementing tunable consistency within the block device allows new high-level storage systems – such as graph stores, new types of filesystems, table stores, databases, etc. – to easily support consistency/performance trade-offs without reimplementing the required mechanisms. We now describe the design and implementation of a block-level StaleStore called Yogurt.

5.2 Yogurt Design

Yogurt is a block-level StaleStore. It exposes a simple, block-level StaleStore API (shown in Table 5.2) that supports timestamped writes, reads and cost estimation. This API is necessary and sufficient for adding StaleStore functionality to a block store; it is analogous to the example key-value StaleStore API shown previously.

Building a block-level StaleStore poses some unique challenges. Applications might prefer to use the standard POSIX-style API for reads and writes to minimize changes to code, and also to use the existing, highly optimized I/O paths from user-space to the block storage driver. Also supporting application-level data abstractions, such as files and key-value pairs, necessitates multiple block accesses. Supporting these require some deviation from the basic StaleStore API. Specifically, Yogurt provides an alternative wrapper API where applications can specify timestamp via explicit control calls (implemented via IOCTLs) and follow those up with POSIX read/write calls.

5.2.1 Block-level StaleStore API

The Yogurt API is simple and matches the generic characteristics of a StaleStore API described in Section 5.1. *ReadVersion(block addr, version)* reads a block corresponding to the version number (specifically, the last written block with a timestamp lower than the supplied version number), and *WriteVersion(block addr, version, data)* writes a block data with the given version number. It is identical to accessing a simple block store, but with an additional version number to read and write the data.

GetCost(block addr, version) is the cost estimation API. The versioned block store computes the integer value to return which can be compared against other *GetCost* calls' results. The smaller the number, the smaller the estimated cost to access it. Depending on the underlying storage settings this number can be configured differently and more details will be presented in Section 5.3.

GetVersionRange(block addr, version) returns a lower and upper bounds of snapshots that contains the specified block intact. An identical block of data can be part of multiple snapshots. This API returns the version number when the block data was last written before the given version number and the version number when the block data is overwritten after the given version number.

5.2.2 Wrapper APIs

As mentioned previously, a standard StaleStore API – consisting of timestamp-augmented versions of the original calls – is problematic for a block store, since it precludes the use of the highly optimized POSIX calls. A second issue for applications is the granularity mismatch between the application and the block store. Application-level consistency is defined at a grain that is either smaller (e.g. small key-value pairs) or larger (e.g. large files) than a single block. In addition, a single access to an application-level construct like a key-value pair or a file often requires multiple accesses at the block level (e.g., one access to look up a key-value index or a filesystem inode; a second access to read the data). If these multiple writes are sent to the StaleStore with different timestamps, the application could potentially access inconsistent snapshots reflecting one write but not the other (e.g., it might see the inode write but not the subsequent data write). Required is a wrapper API that allows applications to use the POSIX calls as well as ensure that inconsistent states of the store cannot be seen.

The answer to both these questions lies in a wrapper API that exposes *Open-Snapshot* and *CloseSnapshot* calls. *OpenSnapshot(version)* opens a snapshot with the specified version number. If the version number is invalid, the operation will fail. The application that opened a snapshot can read one or more blocks within the snapshot using the POSIX read APIs until it closes the snapshot by calling *CloseSnapshot()*.

If the snapshot accessed by the *OpenSnapshot(version)* call is from the past, one cannot directly write new data onto it. To write data, the application supplies a timestamp to *OpenSnapshot()* greater than any the StaleStore has seen before; this opens a writeable snapshot. The application can then write multiple blocks within the snapshot, and then call *CloseSnapshot* to flush the writes out to the store.

Note that the *OpenSnapshot*/*CloseSnapshot* wrapper calls do not provide a full transactional API; they do not handle concurrency control or transactional isolation; the application has to implement its own locking above the wrapper API. However, these calls do provide failure atomicity over the basic StaleStore API.

Under the hood, *OpenSnapshot* simply sets the timestamp to be used for versioned reads and writes. Reads within the snapshot execute via the *ReadVersion* call in the StaleStore API; *CloseSnapshot* flushes writes to the underlying store using the *WriteVersion* call.

5.2.3 Versioned Storage Design

Yogurt implements the block-level StaleStore API over a number of raw block devices, handling I/O requests to multi-versioned data and offering cost estimates for reads. The versioned block store in Yogurt is patterned after Gecko (see Chapter 3 and S3 in Section 2.1.3), which chains a log across multiple devices such that new writes proceed to the tail drive while reads can be served by any drive in the chain. Yogurt maintains a multi-version index over this chained log that maps each logical block address and version number pair to a physical block address on a drive in the chain (in contrast to Gecko, which does not provide multi-version access).

The wrapper layer makes sure that a block is never overwritten with the same version number and a set of writes corresponding to a new snapshot is not exposed to applications other than the one issuing the writes until all writes are persisted in the versioned block store. When *WriteVersion* is called, the versioned block store updates the multi-version index and appends new block data to the log. Similarly, *ReadVersion* simply returns the data corresponding to the address and version pair.

Because the versioned block store sits right on top of block devices, it knows the block device types, characteristics, and how busy each device is. Based on the physical location of each versioned block data, the versioned block store can estimate the cost for accessing a particular version of the block. When *GetCost* API is invoked, the multi-version index is looked up to figure out the physical location of the data, and the access cost is computed based on the storage media speed and the number of queued requests.

5.3 Implementation

Yogurt is implemented as a device mapper, which is a Linux kernel module similar to software RAID and LVM. The wrapper and StaleStore APIs other than the POSIX APIs are implemented as *IOCTL* calls and kernel function calls.

5.3.1 Snapshot Access and Read Mapping

Since modern applications are highly concurrent and serve multiple users, Yogurt should be able to service multiple snapshots to one or many applications. To do this, Yogurt identifies its users using *pid*, which is distinctively given to each thread. When a thread calls *OpenSnapshot*, all read requests from the thread are served from the opened snapshot until the thread calls *CloseSnapshot*. Once a thread is mapped with a snapshot, each read is tagged with the snapshot number and issued via the *ReadVersion* API.

Figure 5.2 shows a logical view of a multi-version index and how a snapshot is constituted. The x-axis is the logical block address and the y-axis is the snapshot version number. Each entry shows a physical block address and a version number corresponding to the logical address. The entries in the same row are the blocks that were written when the snapshot was created. Thus, a snapshot consists of the latest data blocks with version numbers less than or equal to the snapshot's version number.

When the application wishes to access an application-level data item with a certain consistency level, it translates that to a lowest acceptable version number, which we call V_{low}. It then uses the latest snapshot version as V_{up}. Once the application knows the upper and lower bounds V_{up} and V_{low}, it can issue multiple *GetCost* queries within that range. We leave the querying strategy to the application; however, one simple strategy is to assign a query budget Q, and then issue Q *GetCost* requests to a number of versions V_{query} that are uniformly selected between V_{up} and V_{low}:

$$V_{query} = V_{low} + \lfloor((V_{up} - V_{low})/(Q - 1)) \times n\rfloor, \tag{5.1}$$

where $\{n \in \mathbb{Z} | 0 \leq n \leq Q\}$. For example, for upper bound 9, lower bound 5, and querying budget 3, get cost is issued to versions 5, 7, and 9. Depending on the returned costs, the application reads the cheapest version and updates V_{low}, if necessary. If the returned costs are the same for different versions, the application prefers older versions to keep the query range large.

Here, notice that if multiple blocks need to be accessed to read an object (e.g. a file that spans multiple data blocks), the blocks accessed after a block become dependent on the previously accessed block. For example, if an application reads a metadata block, the data block locations are valid only for the snapshots where the metadata block is valid. Say V_{low} and V_{up} were initially set to 0 and 10, respectively by a read semantic and the application read version 1 of logical block 2 in Figure 5.2. Then V_{low} and V_{up} becomes 1 and 5, respectively, which is the range the read block is valid. If version 3 of logical block 7 is read next, the version range becomes $V_{low} = 3$ and $V_{up} = 5$, which is the common range for the two blocks. Similarly, once the application opens a snapshot and reads a block, *GetVersionRange* should be called to update the common V_{low} and V_{up} range while reading the blocks.

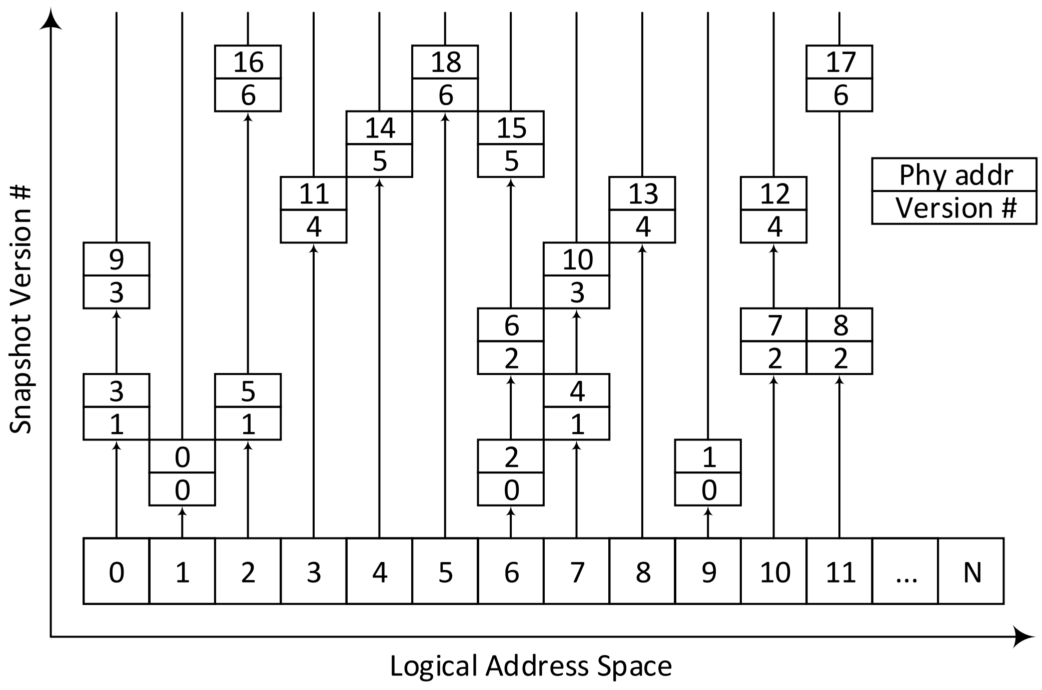


Figure 5.2: Logical illustration of multi-version index and snapshots.

5.3.2 Data Placement

To provide as many read options with different access costs as possible, it can be helpful for Yogurt to save different versions of a same block to different physical storage media. Yogurt uses two data layers to do this: the lower layer consists of the chained log with multiple disks and/or SSDs, and the higher layer is built as a memory cache over the chained log. The memory cache is a LRU based read or read/write cache, where the data written to or read from the bottom layer is cached. When the cache acts as a read/write cache, the data writes through the cache for durability. Perfectly distributing different versions to different block devices is doable in the lower layer, but it can make the versioned block store design complicated and can cause data skew to certain block devices depending on the workload. Instead, the versioned block store uses a simpler approach: data is logged using small segments to each block device in a round robin fashion (e.g. log 16 MB segment to disk 0, log the next 16MB to disk 1, log the next 16MB to disk 2, and then back to disk 0). This results in RAID-0 like throughput behavior, by enabling independent access to each block device.

5.3.3 Read Cost Estimation

Based on the physical storage layer described in the previous subsection, Yogurt returns two-tiered estimated cost. For all *GetCost* calls, the versioned block store first looks up the memory cache. If the searching data block is inside the memory cache, it is always faster to read it from the memory than from either disk or SSD. To indicate this, the cost is returned as a negative value.

If the data block is not in the cache, the cost reflects the number of queued I/Os of the block device containing the block. The versioned block store can trace this using simple counters. For disks and SSDs, precise cost estimation is difficult because the internal states of the block devices are not exposed. Still, there are several known facts that can be applied for estimating the cost: 1) all writes within Yogurt are sequential log appends; 2) mixing random and sequential I/Os within disks results in overall bad performance; 3) mixing reads and writes can penalize read operations in SSDs; and 4) random read latencies of SSDs are orders of magnitude faster than those of disks. Since data blocks are read from a log, we can assume most reads will be accessing physical blocks randomly, and from 1), 2), and 3), separating reads from writes becomes important. So we add more cost to block devices with queued writes and add small cost for queued reads. From 4) we make the cost of reading a disk an order of magnitude more expensive than reading a SSD.

To summarize, there is no cost difference among cached blocks, and cached blocks are the cheapest. SSDs are preferred over disks most of the time, unless there is an order of magnitude more I/Os queued on SSDs. Queued writes are more expensive than reads. Costs are computed as follows:

$$C_{Cache} = -1 \tag{5.2}$$

$$C_{SSD} = C_{rd\_ssd} \times N_r + C_{wr\_ssd} \times N_w \tag{5.3}$$

$$C_{Disk} = C_{rd\_disk} \times N_r + C_{wr\_disk} \times N_w \tag{5.4}$$

where the C variables are the costs of reading from or writing to SSD or disk, and the N variables are the number of queued reads and writes.

5.3.4 A Key-Value Store Example

We describe an example of a key-value store implementation to demonstrate how the Yogurt APIs can be used. The key-value store returns the fastest value of the key while satisfying the consistency constraints of each client.

The key-value store works in the following steps: 1) When a client connects to the key-value store, a session is created for the client and the latest snapshot number of the connected server is used to set up V_{low} values for the key-value pairs depending on the consistency semantics. 2) When the client issues a read to a key-value pair, V_{up} is set to the latest snapshot number of the server and *GetCost* calls are issued to different versions of the metadata block of the key. 3) Based on the returned cost of different versions of the metadata block, the key-value store calls *OpenSnapshot* to read the cheapest version of the block. 4) After the read, *GetVersionRange* is called and V_{low} and V_{up} are updated. 5) Next, the key-value store reads data blocks one by one by calling *GetCost* to versions between V_{low} and V_{up} and going through steps 3) to 4) repeatedly. 6) When the value of the key is completely read, *CloseSnapshot* is called. 7) Finally, depending on the consistency semantics, V_{low} and V_{up} are updated for future reads (e.g. under monotonic-reads, the version of the key-value pair that has been read is recorded in V_{low} and later when the client issues another read, the latest available snapshot number in the server becomes V_{up}).

As shown in the example, it is the responsibility of the application developer to wrap around the access to a single data object using *OpenSnapshot* and *CloseSnapshot*. In addition, *OpenSnapshot* should be repeatedly called within V_{low} and V_{up} range to read multiple blocks so that a data object that spans multiple blocks are read from a consistent snapshot.

5.4 Evaluation

To evaluate the benefit of Yogurt, we implemented a distributed storage service patterned on Pileus [131], where a client accesses a primary server and a secondary server. The primary server always has the latest data and is far away; the secondary server can be stale but is closer to the client. We tested against two variants of this system: one where the distributed service exposed a block API (matching the block store abstraction provided by Yogurt), and a second where it exposed a key-value service to clients. We call these Pileus-Block and Pileus-KV, respectively, and the latter follows the implementation of the example key-value store in the previous section.

The hardware configuration we use under Yogurt is three disks and 256MB memory cache. Data is logged to three disks in round robin in 1GB segments, using a design similar to Gecko's chain logging with smaller segment size. The memory cache can be enabled or disabled as will be described in each experiment.

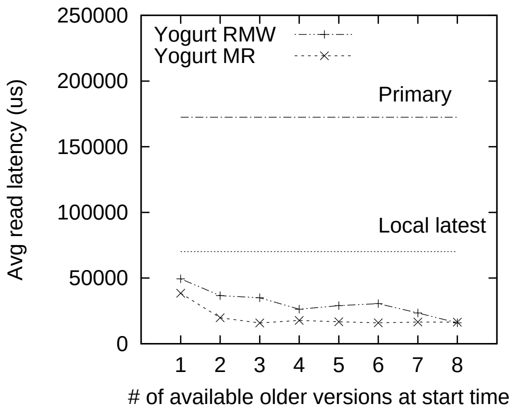
Throughout the evaluation we aim to answer the following questions:

- What is the performance gain we can get by accessing stale data?
- Is there any overhead for accessing older versions?
- How well do real applications run on Yogurt?

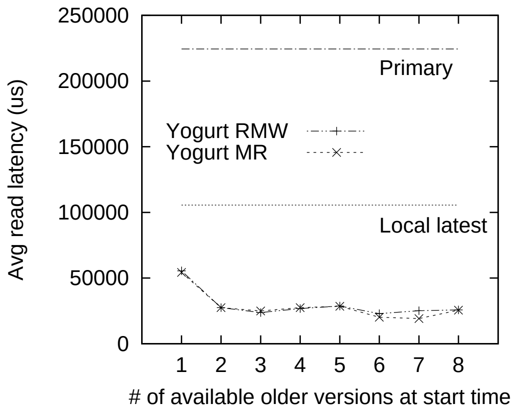
5.4.1 Pileus-Like Block Store

First, we measure the base performance of Yogurt when it is used with a distributed block store, comparing accessing older versions versus accessing only the latest versions. We compare Yogurt against two baseline settings, where the latest versions can be interpreted in different ways: 1) we compare against accessing the latest version within the local server, and 2) against accessing a remote primary server where the globally latest versions reside. We emulate the network latency of accessing the primary server as if it is located across the continental US from the client, delaying the response by 100ms.

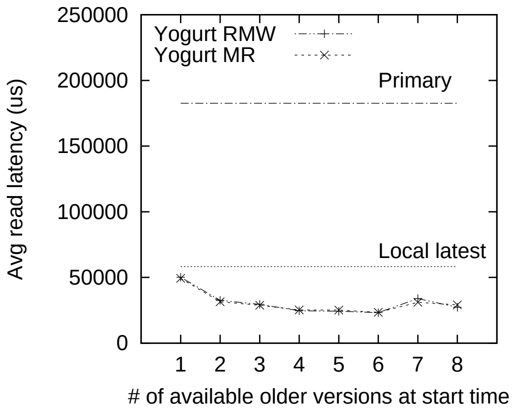
For this evaluation we use two different workloads, uniform random and zipf workload that access 256K blocks. In the local server, we run a thread that



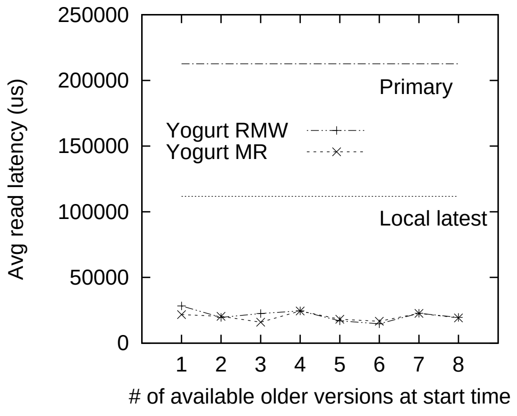
(a) Without memory cache - Uniform random



(b) Without memorycache - Zipf



(c) With memory cache - Uniform random



(d) With memory cache - Zipf

Figure 5.3: Performance of Yogurt under synthetic workloads.

aggressively writes a stream of data coming from the primary node and measured the performance from 8 threads that are reading and writing data in 9 to 1 ratio. The threads run with read-my-writes (RMW) or monotonic reads (MR) consistency guarantees and we start the threads after making *N* versions of data available to them. Figure 5.3 shows the average read latency of 3 runs.

For all cases, accessing the primary server takes the longest as the added network latency is relatively huge and then comes accessing the latest version in the local storage. The latest data in the local server is mostly found in the disk that is writing data and most requests tend to concentrate on this disk.

However, Yogurt can find alternative versions from different disks. The latency quickly drops to 20-25% of accessing the latest version in local storage as the *GetCost* calls enable faster data retrievals. Since there are three disks the best performance is found after being able to access three or more older versions. Monotonic reads semantics show slightly better performance than read-my-writes semantics because writes from the threads that use read-my-writes limit the version range to explore before reading a data that has been written by the thread. Still, being able to explore staleness of one update can provide over 50% latency reduction.

Figures 5.3 (c) and (d) show the cases with memory cache. Although the overall performance of the baseline in (c) and (d) is comparable or better than that of the cases without memory cache (Figures 5.3 (a) and (b)), Yogurt can still return data quicker than the baselines. When there is a cache miss, Yogurt can bring quicker versions as shown with the case without the cache (Figure 5.3 (a) and (b)). Also if a certain version is in the cache (it can be an older version that has been read), Yogurt can reuse the data with better efficiency. For this reason zipf workload that has skewed data access can immediately get large performance gain (Figure 5.3-(d)). This result also shows that Yogurt can take advantage of heterogeneous storage media efficiently.

5.4.2 GetCost Overhead

To access older versions from Yogurt, applications call *GetCost* before every read to find out the lowest cost version. Comparing the cost retrieved from Yogurt is trivial as it is a simple $O(N)$ comparison of numbers. However, *GetCost* function

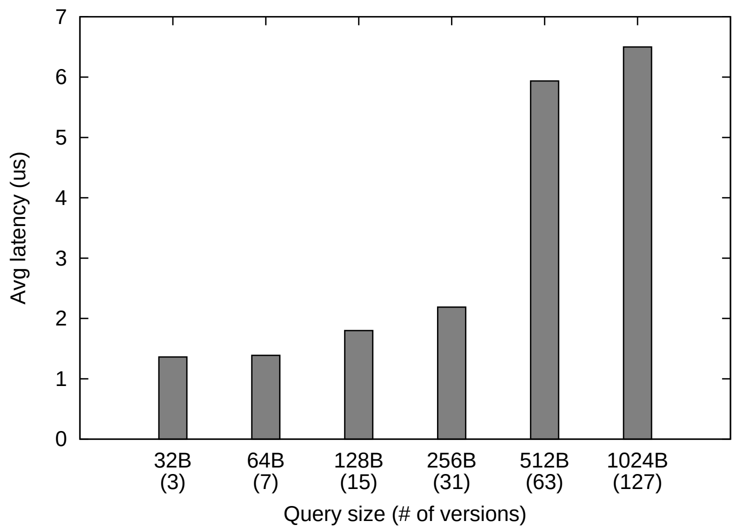


Figure 5.4: GetCost overhead and query size.

call crosses the user space and kernel space boundary and involves copying information which can incur additional latencies.

Figure 5.4 shows the *GetCost* latency of differently sized queries. Larger query size means asking for the cost of larger number of older versions. The larger the query size, the greater the *GetCost* latency. However, considering the read latency of a disk or a SSD which can be tens of microseconds to hundreds of milliseconds, the *GetCost* latency in the figure is very small. All our performance related experiments use 64B queries and results show that we can get far more latency improvements than the 1.4 microsecond overhead.

5.4.3 Pileus-Like Key-Value Store

Pileus-KV uses a persistent hashtable over the Yogurt block address space in order to store variable-sized key-value pairs. We ran YCSB workload *A* which is composed of 50% write and 50% read on the key-value store, choosing keys

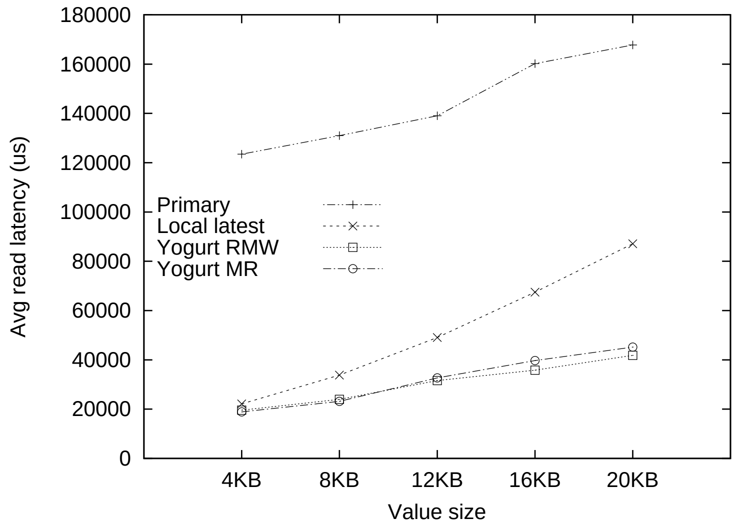


Figure 5.5: Key-value store's read latency and value size.

according to a zipf distribution, and measured the performance. The values of the keys can be partially updated when only part of the value changes. In the experiment, the value size is varied from 4KB to 20KB, which is equivalent to 1 to 5 data blocks. To read or write a key-value pair at least one additional metadata block must be accessed to locate the block that is storing the key. We evaluate Yogurt's capability to access stale data that spans multiple blocks using *GetVersionRange* API with *GetCost* calls.

We used the same server configurations as for the Pileus-like block store and used the memory cache. There are 16 threads accessing the key-value store and a stream of incoming writes from the primary. Figure 5.5 shows the average read latency. As the value size grows to span multiple blocks, Yogurt can provide multiple options for selecting each block. The gap between accessing the latest block from the local storage and accessing older versions grows as the value size gets larger. The key-value store is querying costs every time before it reads, so the overall approach is a simple greedy selection. More sophisticated selection schemes can be proposed to further improve the performance, but the figure shows that for both read-my-writes and monotonic reads semantics greedy selection can already lead to better performance than the baselines.

5.5 Summary

In this chapter, we repurposed a well-known distributed systems principle within the context of a single cloud storage server: storage systems should expose older versions to applications for better performance. This principle enables consistency control within a cloud storage server and provides different isolated views of the storage server to each client. This principle is increasingly relevant as we move toward a post-disk era of storage systems that are often internally multi-versioned and multi-device. Distributed storage services in the cloud can benefit from this principle by pushing relaxed consistency requirements (negotiated between the client and the service) down the stack to the storage subsystem on each server. In the future, we believe that new applications will emerge on a single storage server that can work with weaker consistency guarantees in exchange for better performance.

CHAPTER 6

RELATED WORK

In this dissertation, we explored how to support isolation with regard to performance, transactions, and consistency control in cloud storage systems. In this chapter, we discuss previous work related to our contributions, the techniques on which they build, and alternate or complementary approaches.

6.1 Performance Isolation and Logging

Log-structured storage has a long history starting with the original log-structured filesystem (LFS) [107]. Much of the work on LFS in the 1990s focused on its shortcomings related to garbage collection [115, 82, 116]. Other work, such as Zebra [67], extended LFS-like designs to distributed storage systems. Attempts to distribute logs focused entirely on striping logs over multiple drives, as opposed to the chained-logging design that we investigated in Chapter 3.

Log-structured designs have made a strong comeback in part because of the emergence of flash memory, which requires a log-structured design to minimize wear-out. Not only do individual SSDs layer an address space over a log, but filesystems designed to run over SSDs are often log-structured to minimize the stress on the SSD's internal mapping mechanisms [27]. New log-structured designs have emerged as flash has entered the mainstream; for instance, CORFU [33] uses an off-path sequencer to implement a distributed, shared log over a flash cluster. Another reason for the return of log-structured designs is the increased prevalence of geo-distributed systems, where intrinsic ordering properties of logs provide consistency-related benefits [138].

In addition, performance isolation and contention in data centers have received increasing attention. Lithium [65] uses a single on-disk log structure to support multiple VMs, much as Gecko does (Chapter 3), but it layers this log conventionally over RAID and does not offer any new solutions to the problem of read-write contention. However, the authors of the Lithium paper make two relevant points: first, replicated workloads are even more likely to be write-dominated, and second, the inability of log-structured designs to efficiently service large, sequential reads is unlikely to matter in virtualized settings where such reads are rare due to cross-VM interference. Parallax [85] supports large numbers of virtual disks over a shared block device but focuses on features such as frequent snapshots rather than performance under contention. PARDA [60] is a system that provides fair sharing of a storage array across multiple VMs but does not focus as Gecko does on improving aggregate throughput under contention.

6.2 Transactional Systems

The idea of transactional atomicity for multi-block writes was first proposed in Mime [43], a log-structured storage system that provided atomic multi-sector writes. Over the years, many other projects have proposed block-level or page-level atomicity: the Logical Disk [48] in 1993, Stasis [114] in 2006, TxFlash [101] in 2008, and MARS [44] in 2013. RVM [110] and Rio Vista [79] proposed atomicity over a persistent memory abstraction. All these systems explicitly stopped short of providing full transactional semantics, relying on higher layers to implement isolation. To the best of our knowledge, no existing single-machine system has implemented transactional isolation at the block level, or indeed any concurrency control guarantee beyond linearizability.

On the other hand, distributed filesystems have often relied on the underlying storage layer to provide concurrency control. Boxwood [81], Sinfonia [28], and CalvinFS [132] presented simple network filesystem (NFS) designs that leveraged transactions over distributed implementations of high-level data structures and a shared address space. Transactional isolation has been proposed for shared block storage accessed over a network [29] and for key-value stores [125]. Isotope (Chapter 4) can be viewed as an extension of similar ideas to single-machine, multi-core systems that do not require consensus or distributed rollback protocols. Our single-machine IsoFS implementation has much in common with the Boxwood, Sinfonia, and CalvinFS NFS implementations that ran against clusters of storage servers.

A number of filesystems have been built over a full-fledged database. Inversion [94] is a conventional filesystem built over the POSTGRES database, while Amino [144] is a transactional filesystem (i.e., exposing transactions to users) built over Berkeley DB. WinFS [24] was built over a relational engine derived from SQL Server. This route requires storage system developers to adopt a complex interface – one that does not match or expose the underlying grain of the hardware – in order to obtain benefits such as isolation and atomicity. In contrast, Isotope retains the simple block storage interface while providing isolation and atomicity.

TxOS [100] is a transactional operating system that provides ACID semantics over syscalls, which include file accesses. In contrast, Isotope is largely OS-agnostic and can be ported easily to commodity operating systems or even implemented under the OS as a hardware device. In addition, Isotope supports the easy creation of new systems such as key-value stores and filesystems that run directly over block storage.

Isotope is also related to a large body of work on software transactional memory (STM) [117, 66] systems, which typically provide isolation but not durability or atomicity. Recent work has leveraged new NVRAM technologies to add durability to the software transactional memory (STM) abstraction: Mnemosyne [135] and NV-Heaps [45] with PCM and Hathi [112] with commodity SSDs. In contrast, Isotope aims for transactional secondary storage rather than transactional main-memory.

6.3 Consistency and Performance Trade-Off

The idea of trading off consistency – defined as data freshness – for performance or availability in a distributed system has a rich history [128, 129]. Cloud services ranging from research prototypes such as Pileus [131] and production cloud services such as Amazon SimpleDB [4] offer variable consistency levels. Yogurt (Chapter 5) uses the same trade-off and consistency model within a single cloud storage server context.

6.4 Multi-Versioning

Multi-versioning is the key to handling transactions in Isotope and trading off consistency and performance in Yogurt (Chapter 5. A number of storage systems are multi-versioned mainly for storing and accessing older versions. The systems include WAFL [71] and other filesystems [109, 47, 90], as well as block stores [55, 102]. Also, peer-to-peer storage systems use multi-versioning such as OceanStore [73, 104] and Antiquity [138]. Underlying these systems is research on multiversion data structures [53].

6.5 Smart Block Storage

Although the low-level storage stack, including the block layer, has been kept simple, research on adding more functionality to the block layer has been ongoing. Gecko is inspired heavily by a long line of block-level storage designs, starting with RAID [97]. Such designs typically introduced a layer of indirection at the block level for higher reliability and better performance; for instance, the Logical Disk [48] implemented a log-structured design at the block level for better performance. Log-structured arrays [83] layered a log-structured design over a RAID-5 array. HP AutoRAID [141] switched dynamically between RAID-1 and RAID-5 for hot and cold data, respectively. Petal [76] extended this design to a distributed setting, maintaining an indirection map that could support arbitrary mappings between a logical address space and physical disks. Many systems typically use battery-backed RAM for persisting block-level metadata [111, 83]. While Gecko is similar to these systems in philosophy, it benefits from the availability of commodity flash for achieving persistence, but consequently it must work around the wear-related limitations of flash.

Isotope and Yogurt also fit into block storage systems with rich features, but they add extra block interfaces to support new functionality. Similarly, a number of systems have augmented the block interface [43, 137, 57], modified it [149], and even replaced it with an object-based interface [84].

CHAPTER 7

FUTURE WORK AND CONCLUSION

7.1 Future Work

Isolation in storage systems has been long studied and is continuously explored in the cloud environment. Gecko (Chapter 3), Isotope (Chapter 4), and Yogurt (Chapter 5) contribute to three aspects of isolation – performance, transaction, and consistency control – but there are many future directions and relevant problems that require further research. In this chapter, we review and discuss future research directions.

7.1.1 Hardware Integration

Moving features for isolation below the block layer requires research of further exploration. SSDs have embedded processors and run firmware that carries out complex functionality such as address translation, garbage collection, and caching [27]. Shingled drives [26] that need a data indirection layer have designs similar to those of SSDs. Such modern hardware designs open new possibilities for pushing rich functionality down to the physical block device. Some possible future research directions include the following.

First, operations that are CPU intensive can be offloaded to the block device, thus simplifying the storage stack. As part of this approach, transactions can be pushed down to the physical block device. The block device can offload the caching of uncommitted blocks and computations for comparing transaction conflicts from the host machine. However, additional coordination between block devices for transaction decision-making and committing transactions requires research. A similar approach, which pushes functionality down to a physical block device, can be found on Seagate's new key-value disk drive, which has an Ethernet port and supports key-value interfaces [19]. Seagate's key-value drive facilitates key-value store designs, offloads key-value searching operations from the host machine, and enables bypassing several layers of the storage stack by using Ethernet-based accesses.

Second, a hardware implementation in a physical block device can react quickly to requests. Support for StaleStore APIs is a good candidate for implementation in hardware, because the storage access cost estimation is time sensitive and needs knowledge of the physical block device. Gecko and Isotope rely on flash drives or SSDs to persist metadata and transaction records. Durability guarantees can be best made by the hardware since the storing media's characteristics determine how and when data is persisted.

7.1.2 Support for Distributed Storage Systems

The systems introduced in this dissertation are inside a cloud storage server. Some principles directly apply or extend to distributed storage settings, but some are not immediately usable. For example, transactional APIs of Isotope that are provided from the block layer may not scale in distributed settings. The core idea of handling block level transactions can be applied to a distributed block store, but details such as deciding and aborting transactions should involve network communications among multiple nodes. Network communications can be an overhead in implementing strong isolation guarantees, and operations may need to roll back depending on the implementation. With a centralized controller, coordinating transactions can be easier but the scalability can remain a problem. On the other hand, distributed decision-making can scale well, but it can complicate the design and communication protocols. Large distributed storage systems tend to implement their own transactions with transactional guarantees that are less general and tuned for system-specific needs [125, 29]. A future research direction is to extend the Isotope transactional API to support distributed transactions. The goal would be to enable different semantics under the same API, similar to how Pileus [131] supports different client-centric consistency semantics, but using data-centric consistency models.

One of the challenges that makes distributed transactions difficult is time synchronization: distributed nodes have different clocks and deciding the order of transactions is difficult. There are two approaches to deal with this problem. The first is using logical clocks such as the Lamport clock and vector clock [128]. Following this direction results in systems that are similar to many distributed systems now in common use. However, a recently proposed datacenter time protocol [77] synchronizes physical timestamps at a scale of tens of nanoseconds with bounds using cheap hardware. As another research direction, Isotope could be combined with physically synchronized clocks. We expect such a system to make local decisions for a certain portion of transactions without consulting or with less contact with a centralized decision engine while supporting strong guarantees.

7.1.3 Towards Smarter Block Storage

Finally, a third research direction includes making block storage smarter. Block storage has been treated as very simple, but a great number of features are being integrated similar to the work described in this dissertation. Smarter block storage enables bypassing software stacks, so it can be useful to strip down unnecessary layers in a heavily layered cloud storage system. From the viewpoint of embedded devices that cannot afford heavy layers of software stacks, a smart block store can keep the software stack simple and save power. Block devices are becoming powerful due to advances in hardware technology and there is a need for rethinking the storage stack design. In addition to logging and transactions, deduplication, encryption, data placement for efficient data accesses and fewer defragmentations, and so on can be considered for new features.

Making the block layer smarter requires redesigning other layers such as the filesystem, virtual filesystem, page cache, and even applications at the same time. Gecko, Isotope, and Yogurt demonstrate how new features in the block storage can affect the layers above. Logging and caching approaches inside Gecko and Isotope can influence the caching policy in a page cache and consistency management of Isotope and Yogurt can affect how synchronization works in existing filesystems. The role of each layer in storage stacks will change accordingly, and a great research direction is to investigate how the full storage stack will evolve in the future.

7.2 Conclusion

At the time of writing of this dissertation, cloud storage servers lack support for isolation — performance isolation, transactional isolation and client-centric consistency control. First, cloud storage servers rely on disks that are susceptible to random accesses. A single user executing random I/Os can easily degrade the performance of the storage server and slow down performance for all users. Second, applications in cloud storage servers redundantly implement transactions by repeating the convention of placing complex functionality in high layers of the storage stack. The redundant implementation is a significant burden for developers and different implementations are not compatible with each other. Most cloud applications need transactions or other concurrency control mechanisms, so it is necessary to rethink the storage stack with regard to placing transactions in a more accessible layer. Third, although cloud storage servers have abundant storage resources, making the server as powerful as a distributed system, consistency control within the server had not been investigated. Consistency control within the server provides the opportunity to speed up user access by trading off consistency and performance.

We have explored fundamental approaches to support these types of isolation in cloud storage servers. In particular, we have designed a contention-oblivious disk array based on chained logging to minimize I/O contention and to separate garbage collections from logging operations. This design isolates the performance for users under concurrent storage accesses and also leads to a better performance in general. We designed a block storage system with transactional isolation. The block-level transaction can be ported to any storage stack including and above the block layer and can enable cross-application transactions. Finally, we categorized a new class of systems called StaleStore, which can trade off consistency and performance within a cloud storage server. Using different versions of data and computing the access cost for each version, StaleStore can support various consistency semantics and achieve improved performance. Together, the steps described in this dissertation represent significant progress towards isolation in cloud storage systems.

To validate each approach, we designed, implemented, and evaluated three systems: Gecko, Isotope, and Yogurt. Gecko adds performance isolation to disk-based cloud storage systems. It uses the chained-logging design to address write-write and write-garbage-collection contention and employs a smart SSD cache to minimize read-write contention. Isotope supports transactional isolation from the block layer to enable the easy design of transactional applications. It enables full ACID transactions to be used in any storage stack and across any application with simple API calls. Yogurt controls consistency by using stale data. It can trade off consistency and performance based on client-centric consistency semantics within a server for improved data access latencies.

Cloud storage systems inevitably face challenges regarding isolation as they are foundationally based on resource sharing. As hardware technology advances, the amount of concurrency in the storage system will grow and support for isolation will become more crucial. This dissertation contributes approaches towards isolation in cloud storage environments — performance isolation, transactional isolation, and client-centric consistency control in cloud storage systems.

APPENDIX A

TRANSACTIONS AND ACID PROPERTIES

A transaction is a sequence of operations executed on shared data storage that is carried out in a coherent, reliable, and independent manner even under concurrent accesses to shared data. ACID stands for atomicity, consistency, isolation, and durability, which are the key properties of a transaction. Each item of ACID is used with subtle difference in subfields of computer science, but the database community [37] and this dissertation share the same definitions:

- Atomicity: a transaction should execute completely or not at all, which is known as the all-or-nothing semantics. No effects of a non-completed transaction should be visible to or affect any other transactions. For example, if a system fails while executing a transaction, the state of the system should be recovered in the state before the transaction began.
- Consistency: a transaction should change the data storage's state from a consistent state to another consistent state, which does not violate any integrity constraints. The integrity constraints may vary depending on the system: e.g., a database's primary keys should be unique, a filesystem's inode should point to correct location of data, etc.
- Isolation: when multiple transactions execute concurrently, the transactions should execute as if they were running one at a time in an isolated manner. Isolation defines an ordering constraint that does not violate consistency. For example, strict serializability semantic ensures all transactions to have a total order of execution. Since each transaction preserves consistency, serial execution of transactions will always keep consistency

as well. This, however, should not be confused with atomicity as concurrent atomic transactions can violate consistency: e.g. if two transactions are depositing \$10 to the same bank account that had \$0 balance, both transaction can read \$0 and then add \$10 resulting in a final balance of \$10 instead of \$20. With isolation (e.g. strict serializability), we can make one transaction to execute after another: the latter will always read \$10 and the final balance will be \$20.

- Durability: the updates made by a successfully completed transaction should be stored durably. Even under a system failure, such as power outage, the updates should be preserved. Typically, this means that the updates should be applied in a non-volatile storage media, such as a hard disk drive or a SSD, and not in a volatile storage media, such as a DRAM.

APPENDIX B

CONSISTENCY SEMANTICS

Consistency is a property that is defined and used in many fields – e.g. databases, distributed systems, filesystems, shared or distributed caches and memories in multicore systems, etc. – of computer science. Although the definitions slightly vary, consistency semantics define the restrictions that keep the integrity constraints of a data store: it describes the integrity constraints, the rules to issue operations and the rules to view the results of the operations. In this dissertation, we adopt the definitions of databases and distributed systems communities.

B.1 Database Systems

Consistency semantics of a database are tied with ACID database transactions (Appendix A). The semantics assume atomic and durable operations, and are defined as isolation levels that keep consistencies.

B.1.1 Strict Serializability

Strict serializability is equivalent to being able to schedule transactions in a sequence of time with no overlapping transactions while preserving the order observed by the transaction issuing processes [37]. This is the highest isolation level which leads to the same result as transactions executing one at a time, one after the other on a data storage. However, this means that completely independent transactions can be executed in any order. It can be easily understood as an assumption that the data a transaction is reading will not change by others until the transaction ends. Thus, the design of strict serializability should prevent read-write conflicts of transactions on the same data.

B.1.2 Snapshot Isolation

Snapshot isolation is a guarantee that all reads by a transaction see all updates by transactions that are committed before the transaction started [36]. It means that the reads are served from a consistent snapshot of the data storage that was taken at the beginning of the transaction. A transaction succeeds to commit only if the data that the transaction is trying to update is not updated (and committed) by other transactions after the transaction started. Thus, write-write conflicts of transactions are prohibited.

Snapshot isolation is a weaker guarantee than strict serializability and can cause write skew anomaly. Write skew anomaly occurs when two transactions read two different data from the same snapshot and updates the other data that each transaction read. Both transactions can commit according to the definition of the semantics, but the end result cannot be reached under the strict serializability semantics. For example, two transactions T_1 and T_2 read balances from two bank accounts $B_1 = \$5$ and $B_2 = \$5$, respectively, from the same snapshot. Bank allows withdrawing money as much as the sum of all the accounts from any account. If both T_1 and T_2 tries to withdraw \$10 from B_2 and B_1, respectively, both transactions will succeed. However, the total amount of money withdrawn from the bank will be \$20 resulting in $B_1 = -\$5$ and $B_2 = -\$5$, which the bank did not intend.

B.2 Distributed Systems

Consistency problems of distributed systems stem from the uncertainty of the network and concurrent users accessing multiple machines. Rather than transactions, distributed systems keep consistencies based on a consistency unit (conit), which is like an object or base unit that needs to be maintained consistently [146]. The basic assumptions are that the data is shared and replicated on multiple machines, and the updates of data should be applied to the replicas in a consistent order [1]. In this section, we introduce two categories of consistency models of distributed systems: data-centric and client-centric consistencies [128].

B.2.1 Data-Centric Consistencies

Data-centric consistencies focus on providing a system-wide consistent view of a data storage. Data-centric consistencies assume environments with frequent concurrent updates that require strong ordering guarantees global to the storage.

Sequential Consistency

Sequential consistency was first defined by Leslie Lamport in a multiprocessor context: "The result of any execution is the same as if the operations of all the processors were executed in some sequential order and the operations of

[1]Notice that consistency semantics introduced here should not be confused with distributed transactions, which enforce ACID transactions to multiple machines.

each individual processor appear in this sequence in the order specified by its program [75]." The operations are fetch and store operations, which are reads and writes, and the processors are analogous to distributed nodes in the context of distributed systems. Sequential consistency allows interleaving of read and write operations, but the constraint is that all nodes should see the same interleaving of the operations. Thus, a node does not need to see the latest update of a data at the moment, but should observe an update order that is globally the same.

Causal Consistency

Causal consistency model was presented as a relaxed model of sequential consistency. Causal consistency requires all nodes to agree on the order of causally related effects, but allow concurrent events that are not causally related to be observed in different orders [72]. For example, if a process reads $x = 1$ and writes $y = x + 1$, y's value is causally dependent on x and ordering of the operations should be enforced, but if each $a = 3$ and $b = 5$ are updated by two different processes to random values $a = 1$ and $b = 7$, the updates are not causally related and can be executed in any order. Causal consistency requires keeping track of updates seen by each node to trace the causality. Such tracking results in a dependency graph of orders that should be enforced to all the nodes.

B.2.2 Client-Centric Consistency

Client-centric consistency focuses on guaranteeing consistency specific to individual clients in a distributed system [131, 130]. Client-centric consistency assumes environments with infrequent simultaneous updates (or simultaneous updates that can be easily resolved) and operations mostly performing reads. Writes on each object are assumed to be done in a serial order, which applies the same to replicated nodes. An example of such environment is found in primary backup systems. However, the arrival time of the ordered writes are difficult to predict and this causes consistency semantics to diverge.

Strong Consistency

Strong consistency guarantees that all read operations always see the latest data. Strong consistency is the strongest guarantee as if a client is accessing a non-distributed storage. To achieve this guarantee, however, clients can frequently wait for the updates to be propagated to the nodes that they are accessing.

Eventual Consistency

Eventual consistency has no guarantees for when the updates will arrive at the replicated nodes. Like its name, the semantics only guarantee that the storage system will eventually become consistent. Thus, the client can read any version of data, or any subset of writes that are performed anytime in the past.

Bounded Staleness

Bounded staleness guarantees that the data value that is read by a client is within a certain staleness bound. The staleness bound can be defined in various ways. For example, a time-period bound guarantees that the value returned had been the latest within the given time period and a number-of-updates bound guarantees that the value diverges from the latest data within the defined number of updates.

Monotonic Reads

Monotonic reads guarantee that the value that is read by a client is the same or newer than the value that was last read by the client. Monotonic reads only guarantee per data object (conit) rules, so a client can read a value written at time 100 for object X and then read a value written at time 1 for object Y.

Monotonic Writes

Monotonic writes guarantee that a write to an object by a client should be completed before the same client writes on the same object. It means that if a client writes to an object on one node and then writes to the same object on another node, the former write should be propagated to the latter node, or the latter write has to wait for the former write to be propagated.

Read-My-Writes

Read-my-writes (also known as read-your-writes) guarantee that the value that is read by a client is the same or newer than the value that was last written by the client. Similar to monotonic reads, this is a guarantee per data object (conit).

Writes-Follow-Reads

Writes-follow-reads guarantee that writes by a client on an object following a read by the same client on the same object are executed on the same or newer value than what was read. This is useful for making sure that writes by a client on an object are applied only when the object already exists in a node.

GLOSSARY

Abort An unsuccessful completion of a transaction execution. Updates made within aborted transactions are obliterated from the data store [37].

ACID Atomicity, consistency, isolation, and durability, which are key properties of a transaction [37].

API Application program interface.

Atomicity A property of a transaction. A transaction should execute completely or not at all [37].

Bandwidth The amount of data that can be transferred per unit time. Typically, a bandwidth is indicated as bits per second.

Block device Software or hardware that implements the block abstraction. It consists of linear block address space and corresponding data blocks and provides interfaces to read and write the blocks.

Block store Storage systems that work with block abstractions. Data in the storage is stored as fixed sized blocks in a linear address space and is accessed through the block address.

Bounded staleness One of client-centric consistency semantics, which guarantees that the data read by a client was the latest data within a time bound. The time bound can be replaced with the number of updates [128].

Cache A hardware or software component that stores duplicate data (from a slow storage media) for fast data accesses in the future.

Cache miss A situation where a data item searched from a cache is not found. Cache misses for reads typically involve reading the data item from the lower layers of the storage hierarchy, which can be slow.

Cloud The hardware and software for cloud computing [31].

Cloud computing Both applications delivered as services over the Internet and the system software and hardware in the datacenters that provide those services [31].

Cloud provider A company that provides computing infrastructures, networking services, and applications based on the cloud.

Cloud storage A part of cloud computing services that supports storage functionalities.

Commit A successful completion of a transaction execution. Updates made within committed transactions are permanently applied to the data store [37].

Consistency A property of a transaction. A transaction should change the database's state from a consistent state to another consistent state [37].

Core An independent processing unit within a CPU. A core can be considered as a small CPU.

CPU Central processing unit. An electronic circuit within a computer that carries out basic arithmetic, logical, control, and input/output operations [89].

Data integrity The 'right' condition for a data store, which consistency semantics must preserve. E.g., a database's primary keys should be unique, a filesystem's inode should point to correct location of data, and so on.

Data-centric consistency A class of consistency semantics which focuses on providing a system-wide consistent view of data storage: it involves a form of global ordering of data accesses [128]. Data-centric consistency assumes environments with frequent concurrent updates that require strong ordering guarantees from the viewpoint of the entire storage. Examples of data-centric consistency semantics are causal consistency and sequential consistency.

Deduplication A specialized data compression method that identifies redundant data chunks and keeps a minimal number of copies. Data objects that consist of same data contents use references to point to the same data chunks [86].

Durability A property of a transaction. The updates made by a successfully completed transaction should be stored durably in a stable storage [37].

ECC Error correction code. A special code that is used for detecting and correcting errors in data, for example, by adding redundant bits or parity bits. Error correction code makes data more reliable during a data transfer over a network or a data store in a storage media, which can incur data corruption and bit errors.

End-to-end argument A classical system design principle [108] which mainly explains where to place functions in a system. The paper that proposed this principle uses network system examples and argues that in many cases placing functions in the end application can be better than placing them in the intermediary nodes unless there is compelling reasons for performance or utility.

Eventual consistency One of consistency semantics in distributed systems that guarantees that updates will arrive at the replicated nodes in the future but without any time bounds [128]. Clients of the system can read any versions of data or any subsets of writes that are performed anytime in the past.

Ext2 filesystem The second extended filesystem. Ext2 filesystem is the first commercial grade filesystem for Linux. One of the biggest drawbacks of ext2 filesystem is the unreliability under crash or unclean shutdown of the host system, which is improved by ext3 filesystem [64].

Ext3 filesystem The third extended filesystem. Ext3 filesystem is a Linux filesystem that succeeds Ext2 filesystem. Ext3 filesystem improved reliability under crash or unclean shutdown of the host system by using journaling [64].

Filesystem Filesystem uses file and directory abstractions on top of block devices to control how data is stored and retrieved in a system. It takes care of data placement, indexing, storage space management and so on.

Flash Memory Non-volatile memory that is made of floating gate transistors. It is faster than disks but slower than DRAM. Data is written typically in a 4KB page and a page must be erased before overwriting. Erase granularity is a block which consists of multiple pages. Read, write, and erase are the basic operations, and the read is the fastest and the erase is the slowest operation.

FUSE Filesystem in userspace [6]. A framework that enables userspace design of filesystem.

Garbage collection Memory or storage space management operations that reclaim data objects which are no longer used.

Hybrid drives Persistent data storage drives that use two or more different persistent storage media internally. E.g. a combination of SSD and HDD.

IOCTL A system call for device-specific input/output control operations.

Isolation (1) Encapsulating users or processes in an independent execution environment or keeping them to be less or not at all affected by others [63, 134].

Isolation (2) A property of a transaction. When multiple transactions execute concurrently, the transactions should execute as if they were running one at a time in an isolated manner [37].

Key-value store A data store where each record is associated with a unique key. Storing and retrieving the record requires the corresponding keys and typical interfaces to the data store are put, get and delete.

Latency Time interval between a request and the following response, or a stimulation and the following effect.

Lock A mechanism to enforce limits on accessing a resource. For example, when multiple processes share a resource (e.g. a piece of data in memory) and a process locks the resource, depending on the type of the lock, the process can get exclusive access to the resource. Locks are commonly used to arbitrate multiple user requests on a shared resource to guarantee safe and consistent accesses.

Log A data structure that appends all data writes sequentially [107].

Log-structured merge tree A data structure that consists of multiple levels of trees [95]. Typically, the high-level tree exists in memory and the low-level tree exists on disk. Each tree maintains a set of sorted data. Once the high-level three reaches certain threshold size the nodes in the high-level tree are evicted and merged with the nodes in the low-level tree and logged to the disk. The multi-level structure makes it suitable for frequent write operations and merging and logging in low-levels make it suitable for range search of data.

MLC Multi-level cell. Memory cells that can have more than two states. Each cell can store more than one bit.

Monotonic reads One of client-centric consistency semantics, which guarantees that the value of a data read by a client is the same or newer than the previously read value of the same data by the same client [128].

Multi-version concurrency control A concurrency control method which uses multiple versions of data/snapshots instead of locks so that multiple users can concurrently access the data [37].

NVRAM Non-volatile random access memory. Flash memories and phase-change memories fall into this category.

Optimistic concurrency control A concurrency control method which assumes that there are no transactions that have conflicting data accesses [37]. It lets any transactions access any data. At the end of a transaction execution, the transaction is tested whether it has any conflicting data accesses with other transactions. If there is no conflict, the transaction commits, otherwise, it aborts.

Performance isolation A property that minimizes noticeable contention of resources and access time delays in systems to make users unaware of each other's behaviors [134, 63].

Pessimistic concurrency control A concurrency control method which assumes that there are always transactions that have conflicting data accesses [37]. It uses locks to prevent transactions from executing prohibited data accesses.

Phase change memory A type of non-volatile random access memory which uses heat and crystallization to encode bits [143]. It has a faster data access latency but a lower circuit density than a flash memory.

POSIX Portable operating system interface. A set of standards for compatibility of operating systems.

PRAM See phase change memory.

RAID Redundant array of inexpensive/independent disks [97]. It ties together multiple disks, erasure codes the data, and stripes the data and the erasure code to the disks for better performance and reliability. There are different levels of RAID, which defines the how data is encoded and placed.

RAID-0 One of RAID levels that evenly stripes data to disks with no parity bits.

RAID-1 One of RAID levels that mirrors (makes exact same copy) data to disks.

RAID-10 A combination of RAID-0 and RAID-1, where the data is first striped to groups of disks and mirrored within each group.

RAID-4 One of RAID levels that evenly stripes data to disks with a dedicated parity disk. Can tolerate one disk failure.

RAID-5 One of RAID levels that evenly stripes data to disks with distributed parity all over the disks. Improved version of RAID-4 by distributing concentrated parity writes in a dedicated parity disk. Can tolerate one disk failure.

RAID-6 One of RAID levels that evenly stripes data to disks with two parity blocks distributed all over the disks. Can tolerate two disk failures.

RAM Random access memory. A form of computer data storage typically built out of transistors. The memory cell that stores the data is an electronic circuit that can be quickly accessed but loses data when the power goes out. The random access memory is used as a cache for CPUs (static random access memory) and as a main memory (dynamic random access memory) for a computer depending on the type.

Random I/O Accesses to data items in nonconsecutive addresses. Random I/O in hard disk drives requires seek operations.

Read-my-writes One of client-centric consistency semantics, which guarantees that the value of a data read by a client is the same or newer than the previously written value of the same data by the same client [128].

RPM Abbreviation for revolutions per minute. A measurement unit that is used for indicating the speed of disk rotation in hard disk drives.

Seek A disk arm movement between the inner and the outer track of a platter.

Sequential I/O Accesses to data items in consecutive addresses. Sequential I/O in hard disk drives does not incur seek operations.

Skiplist Multiple layered list that enables fast searching. The lowest layer is an ordinary linked list and as the layer goes higher the list becomes sparser: i.e. the higher layer list includes a fewer number of entries by skipping a larger number of entries at the lowest layer. Thus, similar to a tree structure, search happens from the highest layer in a coarse-grained manner and then the search range becomes smaller as the operation moves towards the lowest layer.

SLC Single level cell. Memory cells that have only two states. Each cell can store only one bit.

Snapshot A state of the storage system at a particular time point.

Snapshot isolation One of transactional isolation semantics which guarantees that all reads by a transaction see all updates by transactions that had successfully completed before the transaction started [36].

SSD Solid State Drive (Disk). Persistent block device made of flash memory.

Strict serializability One of transactional isolation semantics which is equivalent to scheduling concurrent transactions sequentially one after another with no overlapping transactions while preserving the order observed by the transaction issuing processes [37].

Transaction A sequence of operations carried out in a reliable, independent and consistent way on a shared storage [37].

Virtual machine (VM) An emulated computer machine that runs on top of virtualized hardware devices. The devices, which are in reality shared hardware resources, are created by hypervisor software [34].

Virtualization Creating and managing multiple virtual machines on a physical machine. Virtualization is enabled by hypervisor software, which isolates virtual machines and multiplexes shared hardware resources to them [34].

Weak consistency Refers to consistency models weaker than sequential consistency.

Write skew anomaly An anomalous state that is reachable when two transactions read overlapping values, make disjoint updates to the values, and concurrently commit. This is called anomaly as the state is unreachable under serializablily semantics [36].

BIBLIOGRAPHY

[1] Amazon elastic block store. https://aws.amazon.com/ebs/.

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

[3] Amazon S3 - two trillion objects, 1.1 million requests / second. https://aws.amazon.com/blogs/aws/amazon-s3-two-trillion-objects-11-million-requests-second/.

[4] Amazon SimpleDB. https://aws.amazon.com/simpledb/.

[5] fcntl man page. http://man7.org/linux/man-pages/man2/fcntl.2.html.

[6] Filesystem in userspace (FUSE). https://github.com/libfuse/libfuse.

[7] Fusion-io. www.fusionio.com.

[8] Google cloud storage. https://cloud.google.com/storage/.

[9] IOZone filesystem benchmark. http://www.iozone.org.

[10] LevelDB. https://github.com/google/leveldb.

[11] LevelDB benchmarks. http://leveldb.googlecode.com/svn/trunk/doc/benchmark.html.

[12] Linux device mapper documentation. https://www.kernel.org/doc/Documentation/device-mapper/.

[13] mdadm man page.

[14] Microsoft Azure storage. https://azure.microsoft.com/en-us/services/storage/.

[15] Next generation EMC: Lead your storage transformation. https://www.emc.com/campaign/global/forum2013/pdf/ch-storage-next-generation-emc.pdf.

[16] Roundup of cloud computing forecasts and market estimates, 2016. http://www.forbes.com/sites/louiscolumbus/2016/03/13/roundup-of-cloud-computing-forecasts-and-market-estimates-2016.

[17] SanDisk Fusion-io atomic multi-block writes. `http://www.sandisk.com/assets/docs/accelerate-myql-open-source-databases-with-sandisk-nvmfs-and-fus` `pdf`.

[18] SanDisk Fusion-io auto-commit memory. `http://web.sandisk.com/assets/white-papers/MySQL_High-Speed_Transaction_Logging.pdf`.

[19] Seagate Kinetic open storage platform. `http://www.seagate.com/solutions/cloud/data-center-cloud/platforms/`.

[20] SNIA IOTTA repository. `http://iotta.snia.org/`.

[21] Storage spaces overview. `https://technet.microsoft.com/en-us/library/hh831739(v=ws.11).aspx`.

[22] US federal government in the cloud. https://aws.amazon.com/federal/.

[23] Windows Azure storage - 4 trillion objects and counting. https://azure.microsoft.com/en-us/blog/windows-azure-storage-4-trillion-objects-and-counting/.

[24] WinFS. `http://blogs.msdn.com/b/winfs/`.

[25] Anant Agarwal and Markus Levy. The kill rule for multicore. In *Design Automation Conference*, 2007.

[26] Abutalib Aghayev and Peter Desnoyers. Skylight a window on shingled disk operation. In *USENIX Conference on File and Storage Technologies*, 2015.

[27] N. Agrawal, V. Prabhakaran, T. Wobber, J.D. Davis, M. Manasse, and R. Panigrahy. Design tradeoffs for SSD performance. In *USENIX Annual Technical Conference*, 2008.

[28] Marcos K Aguilera, Arif Merchant, Mehul Shah, Alistair Veitch, and Christos Karamanolis. Sinfonia: a new paradigm for building scalable

distributed systems. *ACM SIGOPS Operating Systems Review*, 41(6):159–174, 2007.

[29] Khalil Amiri, Garth A Gibson, and Richard Golding. Highly concurrent shared storage. In *IEEE International Conference on Distributed Computing Systems*, 2000.

[30] David G. Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. FAWN: A fast array of wimpy nodes. In *ACM Symposium on Operating Systems Principles*, 2009.

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

[32] Anirudh Badam and Vivek S Pai. SSDAlloc: hybrid SSD/RAM memory management made easy. In *USENIX Symposium on Networked Systems Design and Implementation*, 2011.

[33] M. Balakrishnan, D. Malkhi, V. Prabhakaran, T. Wobber, M. Wei, and J. Davis. CORFU: A shared log design for flash clusters. In *USENIX Symposium on Networked Systems Design and Implementation*, 2012.

[34] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *ACM Symposium on Operating Systems Principles*, 2003.

[35] Salman A. Baset. Cloud SLAs: Present and future. *ACM SIGOPS Operating Systems Review*, 46(2):57–66, 2012.

[36] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ANSI SQL isolation levels. *ACM SIGMOD Record*, 24(2):1–10, 1995.

[37] Philip Bernstein and Eric Newcomer. *Principles of Transaction Processing*. Morgan Kaufmann Publishers, 2 edition, 2009.

[38] Philip A Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency control and recovery in database systems*. Addison-Wesley Longman Publishing, 1987.

[39] Andrew Birrell, Michael Isard, Chuck Thacker, and Ted Wobber. A design for high-performance flash disks. *ACM SIGOPS Operating Systems Review*, 41(2):88–93, 2007.

[40] Daniel Bovet and Marco Cesati. *Understanding The Linux Kernel*. Oreilly & Associates Inc, 2005.

[41] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. Web caching and zipf-like distributions: Evidence and implications. In *IEEE International Conference on Computer Communications*, 1999.

[42] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, et al. Windows Azure storage: A highly available cloud storage service with strong consistency. In *ACM Symposium on Operating Systems Principles*, 2011.

[43] Chia Chao, Robert English, David Jacobson, Alexander Stepanov, and John Wilkes. Mime: a high performance parallel storage device with strong recovery guarantees. Technical Report HPL-CSP-92-9, Hewlett-Packard Laboratories, 1992.

[44] Joel Coburn, Trevor Bunker, Rajesh K Gupta, and Steven Swanson. From ARIES to MARS: Reengineering transaction management for next-generation, solid-state drives. In *ACM Symposium on Operating Systems Principles*, 2013.

[45] Joel Coburn, Adrian M Caulfield, Ameen Akel, Laura M Grupp, Rajesh K Gupta, Ranjit Jhala, and Steven Swanson. NV-Heaps: making persistent objects fast and safe with next-generation, non-volatile memories. *ACM SIGARCH Computer Architecture News*, 39(1):105–118, 2011.

[46] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *ACM Symposium on Cloud Computing*, 2010.

[47] Brian Cornell, Peter A Dinda, and Fabián E Bustamante. Wayback: A user-level versioning file system for linux. In *USENIX Annual Technical Conference*, 2004.

[48] Wiebren De Jonge, M Frans Kaashoek, and Wilson C Hsieh. The logical disk: A new approach to improving file systems. *ACM SIGOPS Operating Systems Review*, 27(5):15–28, 1993.

[49] Dell Inc. *Dell PowerEdge 2450 Systems Installation and trouble shooting guide*, November 1999.

[50] Dell Inc. *Dell PowerEdge 2850 Server*, September 2005.

[51] Dell Inc. *Dell PowerEdge R930*, May 2016.

[52] David J DeWitt, Randy H Katz, Frank Olken, Leonard D Shapiro, Michael R Stonebraker, and David A. Wood. Implementation techniques for main memory database systems. In *ACM SIGMOD International Conference on Management of Data*, 1984.

[53] James R Driscoll, Neil Sarnak, Daniel Dominic Sleator, and Robert Endre Tarjan. Making data structures persistent. In *ACM symposium on Theory of computing*, 1986.

[54] Bin Fan, David G Andersen, and Michael Kaminsky. MemC3: Compact and concurrent memcache with dumber caching and smarter hashing. In *USENIX Symposium on Networked Systems Design and Implementation*, 2013.

[55] Michail Flouris and Angelos Bilas. Clotho: Transparent data versioning at the block i/o level. In *International Conference on Massive Storage Systems and Technology*, 2004.

[56] Richard F. Freitas, Joe Slember, Wayne Sawdon, and Lawrence Chiu. GPFS scans 10 billion files in 43 minutes. Technical Report RJ10484 (A1107-011), IBM, July 2011.

[57] Gregory R Ganger. *Blurring the line between OSes and storage devices*. School of Computer Science, Carnegie Mellon University, 2001.

[58] Laura M. Grupp, John D. Davis, and Steven Swanson. The bleak future of nand flash memory. In *USENIX Conference on File and Stroage Technologies*, 2012.

[59] Rachid Guerraoui and Michal Kapalka. On the correctness of transactional memory. In *ACM SIGPLAN Symposium on Principles and practice of parallel programming*, 2008.

[60] A. Gulati, I. Ahmad, and C.A. Waldspurger. PARDA: Proportional allocation of resources for distributed storage access. In *USENIX Conference on File and Storage Technologies*, 2009.

[61] A. Gulati, C. Kumar, and I. Ahmad. Storage workload characterization and consolidation in virtualized environments. In *Workshop on Virtualization Performance: Analysis, Characterization, and Tools*, 2009.

[62] A. Gulati, G. Shanmuganathan, I. Ahmad, C. Waldspurger, and M. Uysal. Pesto: Online storage performance management in virtualized datacenters. In *ACM Symposium on Cloud Computing*, 2011.

[63] Diwaker Gupta, Ludmila Cherkasova, Rob Gardner, and Amin Vahdat. Enforcing performance isolation across virtual machines in xen. In *ACM/IFIP/USENIX Middleware Conference*, 2006.

[64] William Von Hagen. *Linux Filesystems*. Sams, 2001.

[65] Jacob Gorm Hansen and Eric Jul. Lithium: Virtual machine storage for the cloud. In *ACM Symposium on Cloud Computing*, 2010.

[66] Tim Harris, James Larus, and Ravi Rajwar. *Transactional Memory*. Morgan and Claypool Publishers, 2010.

[67] John H. Hartman and John K. Ousterhout. The zebra striped network file system. *ACM Transactions on Computer Systems*, 13(3):274–310, 1995.

[68] M. P. Herlihy and J. M. Wing. Axioms for concurrent objects. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1987.

[69] HGST. *Ultrastar C10K1800*, 2015.

[70] HGST. *Ultrastar He10*, 2016.

[71] Dave Hitz, James Lau, and Michael A Malcolm. File system design for an nfs file server appliance. In *USENIX Winter Technical Conference*, 1994.

[72] P. W. Hutto and M. Ahamad. Slow memory: weakening consistency to enhance concurrency in distributed shared memories. In *International Conference on Distributed Computing Systems*, 1990.

[73] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Chris Wells, and Ben Zhao. OceanStore: An architecture for

global-scale persistent storage. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2000.

[74] Hsiang-Tsung Kung and John T Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, 6(2):213–226, 1981.

[75] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 100(9):690–691, 1979.

[76] E. K. Lee and C. A. Thekkath. Petal: Distributed virtual disks. *ACM SIGOPS Operating Systems Review*, 30(5):84–92, 1996.

[77] Ki Suh Lee, Han Wang, Vishal Shrivastav, and Hakim Weatherspoon. Globally synchronized time via datacenter networks. In *ACM SIGCOMM Conference on Data Communication*, 2016.

[78] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't settle for eventual: Scalable causal consistency for wide-area storage with cops. In *ACM Symposium on Operating Systems Principles*, 2011.

[79] David E Lowell and Peter M Chen. Free transactions with rio vista. *ACM SIGOPS Operating Systems Review*, 31(5):92–101, 1997.

[80] Lanyue Lu, Yupu Zhang, Thanh Do, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Physical disentanglement in a container-based file system. In *USENIX Symposium on Operating Systems Design and Implementation*, 2014.

[81] John MacCormick, Nick Murphy, Marc Najork, Chandramohan A Thekkath, and Lidong Zhou. Boxwood: Abstractions as the foundation for storage infrastructure. In *USENIX Symposium on Operating Systems Design and Implementation*, 2004.

[82] J.N. Matthews, D. Roselli, A.M. Costello, R.Y. Wang, and T.E. Anderson. Improving the performance of log-structured file systems with adaptive methods. In *ACM Symposium on Operating System Principles*, 1997.

[83] J. Menon. A performance comparison of raid-5 and log-structured arrays.

In *IEEE International Symposium on High Performance Distributed Computing*, 1995.

[84] Mike Mesnier, Gregory R Ganger, and Erik Riedel. Object-based storage. *IEEE Communications Magazine*, 41(8):84–90, 2003.

[85] Dutch T Meyer, Gitika Aggarwal, Brendan Cully, Geoffrey Lefebvre, Michael J Feeley, Norman C Hutchinson, and Andrew Warfield. Parallax: virtual disks for virtual machines. *ACM SIGOPS Operating Systems Review*, 42(4):41–54, 2008.

[86] Dutch T. Meyer and William J. Bolosky. A study of practical deduplication. In *USENIX Conference on File and Stroage Technologies*, 2011.

[87] C Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems*, 17(1):94–162, 1992.

[88] Katherine F. Moore and Dan Grossman. High-level small-step operational semantics for transactions. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 51–62, 2008.

[89] Scott Mueller. *Upgrading and Repairing Pcs, Eighteenth Edition*. Que Corp., 22 edition, 2015.

[90] Kiran-Kumar Muniswamy-Reddy, Charles P Wright, Andrew Himmer, and Erez Zadok. A versatile and user-oriented versioning file system. In *USENIX Conference on File and Storage Technologies*, 2004.

[91] D. Narayanan, E. Thereska, A. Donnelly, S. Elnikety, and A. Rowstron. Migrating server storage to SSDs: Analysis of tradeoffs. In *European Conference on Computer Systems*, 2009.

[92] D. Nellans, M. Zappe, J. Axboe, and D. Flynn. ptrim ()+ exists (): Exposing new FTL primitives to applications. In *Non-Volatile Memories Workshop*, 2011.

[93] Edmund B Nightingale, Jeremy Elson, Jinliang Fan, Owen Hofmann, Jon Howell, and Yutaka Suzue. Flat datacenter storage. In *USENIX Symposium on Operating Systems Design and Implementation*, 2012.

[94] Michael A Olson. The design and implementation of the inversion file system. In *USENIX Winter Technical Conference*, 1993.

[95] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The log-structured merge-tree (LSM-tree). *Acta Informatica*, 33(4):351–385, June 1996.

[96] T. M. Oslon. Disk array performance in a random IO environment. *ACM SIGARCH Computer Architecture News*, 17(5):71–77, 1989.

[97] David A. Patterson, Garth Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (RAID). In *ACM SIGMOD International Conference on Management of Data*, pages 109–116, 1988.

[98] Avery Pennarun. Everything you never wanted to know about file locking. `http://apenwarr.ca/log/?m=201012#13`.

[99] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17(7):412–421, 1974.

[100] Donald E Porter, Owen S Hofmann, Christopher J Rossbach, Alexander Benn, and Emmett Witchel. Operating system transactions. In *ACM Symposium on Operating Systems Principles*, 2009.

[101] Vijayan Prabhakaran, Thomas L Rodeheffer, and Lidong Zhou. Transactional flash. In *USENIX Symposium on Operating Systems Design and Implementation*, 2008.

[102] Sean Quinlan and Sean Dorward. Venti: A new approach to archival storage. In *USENIX Conference on File and Storage Technologies*, 2002.

[103] Colin Reid, Philip A Bernstein, Ming Wu, and Xinhao Yuan. Optimistic concurrency control by melding trees. *Proceedings of the VLDB Endowment*, 4(11), 2011.

[104] Sean Rhea, Patrick Eaton, Dennis Geels, Hakim Weatherspoon, Ben Zhao, and John Kubiatowicz. Pond: The OceanStore prototype. In *USENIX Conference on File and Storage Technologies*, 2003.

[105] RightScale Inc. State of the cloud report, 2016.

[106] David Roberts, Taeho Kgil, and Trevor Mudge. Integrating NAND flash devices onto servers. *Communications of the ACM*, 52(4):98–103, 2009.

[107] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transaction on Computer Systems*, 10(1):26–52, 1992.

[108] Jerome H Saltzer, David P Reed, and David D Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288, 1984.

[109] Douglas S Santry, Michael J Feeley, Norman C Hutchinson, Alistair C Veitch, Ross W Carton, and Jacob Ofir. Deciding when to forget in the elephant file system. *ACM SIGOPS Operating Systems Review*, 33(5):110–123, 1999.

[110] Mahadev Satyanarayanan, Henry H Mashburn, Puneet Kumar, David C Steere, and James J Kistler. Lightweight recoverable virtual memory. *ACM Transactions on Computer Systems*, 12(1):33–57, 1994.

[111] S. Savage and J. Wilkes. AFRAID: A frequently redundant array of independent disks. In *USENIX Annual Technical Conference*, 1996.

[112] Mohit Saxena, Mehul A Shah, Stavros Harizopoulos, Michael M Swift, and Arif Merchant. Hathi: durable transactions for memory using flash. In *International Workshop on Data Management on New Hardware*, pages 33–38, 2012.

[113] Mohit Saxena, Michael M Swift, and Yiying Zhang. FlashTier: a lightweight, consistent and durable storage cache. In *European Conference on Computer Systems*, 2012.

[114] Russell Sears and Eric Brewer. Stasis: Flexible transactional storage. In *USENIX Symposium on Operating Systems Design and Implementation*, 2006.

[115] M. Seltzer, K. Bostic, M.K. McKusick, and C. Staelin. An implementation of a log-structured file system for UNIX. In *USENIX Winter Technical Conference*, 1993.

[116] M. Seltzer, K.A. Smith, H. Balakrishnan, J. Chang, S. McMains, and V. Padmanabhan. File system logging versus clustering: A performance comparison. In *USENIX Annual Technical Conference*, 1995.

[117] Nir Shavit and Dan Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, 1997.

[118] Ji-Yong Shin, Mahesh Balakrishnan, Lakshmi Ganesh, Tudor Marian, and Hakim Weatherspoon. Gecko: A contension-oblivious design for cloud storage. In *USENIX Workshop on Hot Topics in Storage and File Systems*, 2012.

[119] Ji-Yong Shin, Mahesh Balakrishnan, Tudor Marian, and Hakim Weatherspoon. Gecko: Contention-oblivious disk arrays for cloud storage. In *USENIX Conference on File and Storage Technologies*, 2013.

[120] Ji-Yong Shin, Zeng-Lin Xia, Ning-Yi Xu, Rui Gao, Xiong-Fei Cai, Seungryoul Maeng, and Feng-Hsiung Hsu. FTL design exploration in reconfigurable high-performance SSD for server applications. In *Proceedings of International Conference on Supercomputing*, 2009.

[121] David Shue, Michael J. Freedman, and Anees Shaikh. Performance isolation and fairness for multi-tenant cloud storage. In *USENIX Symposium on Operating Systems Design and Implementation*, 2012.

[122] Abraham Silberschatz, Henry Korth, and S. Sudarshan. *Database Systems Concepts*. McGraw-Hill, Inc., 5 edition, 2006.

[123] Dimitris Skourtis, Dimitris Achlioptas, Noah Watkins, Carlos Maltzahn, and Scott Brandt. Flash on rails: consistent flash performance through redundancy. In *USENIX Annual Technical Conference*, 2014.

[124] G. Soundararajan, V. Prabhakaran, M. Balakrishnan, and T. Wobber. Extending SSD lifetimes with disk-based write caches. In *Proceedings of USENIX Conference on File and Storage Technologies*, 2010.

[125] Yair Sovran, Russell Power, Marcos K Aguilera, and Jinyang Li. Transactional storage for geo-replicated systems. In *ACM Symposium on Operating Systems Principles*, 2011.

[126] Lex Stein. Stupid file systems are better. In *Workshop on Hot Topics in Operating Systems*, 2005.

[127] Michael Stonebraker. Implementation of integrity constraints and views by query modification. In *ACM SIGMOD International Conference on Management of Data*, 1975.

[128] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems: Principles and Paradigms (2Nd Edition)*. Prentice-Hall, Inc., 2006.

[129] D. B. Terry, M. M. Theimer, Karin Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. In *ACM Symposium on Operating Systems Principles*, 1995.

[130] Doug Terry. Replicated data consistency explained through baseball. *Communications of the ACM*, 56(12):82–89, 2013.

[131] Douglas B Terry, Vijayan Prabhakaran, Ramakrishna Kotla, Mahesh Balakrishnan, Marcos K Aguilera, and Hussam Abu-Libdeh. Consistency-based service level agreements for cloud storage. In *ACM Symposium on Operating Systems Principles*, 2013.

[132] Alexander Thomson and Daniel J. Abadi. CalvinFS: Consistent wan replication and scalable metadata management for distributed file systems. In *USENIX Conference on File and Storage Technologies*, 2015.

[133] Robbert Van Renesse and Fred B Schneider. Chain replication for supporting high throughput and availability. In *USENIX Symposium on Operating Systems Design and Implementation*, 2004.

[134] Ben Verghese, Anoop Gupta, and Mendel Rosenblum. Performance isolation: Sharing and isolation in shared-memory multiprocessors. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 1998.

[135] Haris Volos, Andres Jaan Tack, and Michael M Swift. Mnemosyne: Lightweight persistent memory. *ACM SIGARCH Computer Architecture News*, 39(1):91–104, 2011.

[136] M. Wang, K. Au, A. Ailamaki, A. Brockwell, C. Faloutsos, and G. R. Ganger. Storage device performance prediction with CART models. In *IEEE Modeling, Analysis, and Simulation On Computer and Telecommunication Systems Conference*, 2004.

[137] Randolph Y Wang, Thomas E Anderson, and David A Patterson. Virtual log based file systems for a programmable disk. *ACM SIGOPS Operating Systems Review*, 33:29–44, 1998.

[138] H. Weatherspoon, P. Eaton, B.G. Chun, and J. Kubiatowicz. Antiquity: Exploiting a Secure Log for Wide-Area Distributed Storage. *ACM SIGOPS Operating Systems Review*, 41(3):371–384, 2007.

[139] Western Digital. *WD Caviar Desktop Hard Drives*, 2007.

[140] Western Digital. Specifications for the WD Caviar AC22500. `http://www.wdc.com/en/products/legacy/Legacy.asp?Model=AC22500`, 2016.

[141] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan. The HP AutoRAID hierarchical storage system. *ACM Transactions on Computer Systems*, 14(1):108–136, 1996.

[142] Dan Williams, Hani Jamjoom, Yew-Huey Liu, and Hakim Weatherspoon. Overdriver: Handling memory overload in an oversubscribed cloud. In *ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, 2011.

[143] H. S. P. Wong, S. Raoux, S. Kim, J. Liang, J. P. Reifenberg, B. Rajendran, M. Asheghi, and K. E. Goodson. Phase change memory. *Proceedings of the IEEE*, 98(12):2201–2227, 2010.

[144] Charles P Wright, Richard Spillane, Gopalan Sivathanu, and Erez Zadok. Extending ACID semantics to the file system. *ACM Transactions on Storage*, 3(2):4, 2007.

[145] Guanying Wu and Xubin He. Delta-FTL: Improving SSD lifetime via exploiting content locality. In *European Conference on Computer Systems*, 2012.

[146] Haifeng Yu and Amin Vahdat. Design and evaluation of a continuous consistency model for replicated services. In *USENIX Symposium on Operating Systems Design and Implementation*, 2000.

[147] Xiang Yu, Benjamin Gum, Yuqun Chen, Randolph Y Wang, Kai Li, Arvind Krishnamurthy, and Thomas E Anderson. Trading capacity for performance in a disk array. In *USENIX Symposium on Operating System Design and Implementation*, 2000.

[148] Yinqian Zhang, Ari Juels, Alina Oprea, and Michael K. Reiter. Homealone: Co-residency detection in the cloud via side-channel analysis. In *IEEE Symposium on Security and Privacy*, 2011.

[149] Yiying Zhang, Leo Prasath Arulraj, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. De-indirection for flash-based ssds with nameless writes. In *USENIX Conference on File and Storage Technologies*, 2012.