Non-Invasive I/O Classification Techniques and Applications

By

Leo Prasath Arulraj

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2018

Date of final oral examination: December 13th 2017

The dissertation is approved by the following members of the Final Oral Committee:

Andrea C. Arpaci-Dusseau, Professor, Computer Sciences

Remzi H. Arpaci-Dusseau, Professor, Computer Sciences

Michael M. Swift, Professor, Computer Sciences

Aditya Akella, Professor, Computer Sciences

Jonathan T. Eckhardt, Associate Professor, Management & Human Resources

*To my family*

# Acknowledgments

I am extremely thankful and indebted to my advisors Profs. Andrea and Remzi Arpaci-Dusseau for accepting me as their student and for their excellent guidance throughout my Ph.D. program. I always admired their elegant solutions to complex problems. Often times, I used to go to our weekly meetings without clarity and talking to them just for a few minutes added clarity to my thoughts. They taught me the essential skill of converting a complex piece of technology that we built into an engaging high level presentation that conveys all the necessary points but yet does not bore the listener. When the papers we submitted to conferences were rejected, they taught me how to handle rejections with their invaluable guidance on choosing the most important feedback from the reviews that can be addressed in the available time. I have always been amazed by their clarity of thought process in steering a project in the right direction.

Prof. Andrea's detailed comments, technical as well as grammatical corrections, and insightful questions have always been instrumental in adding clarity and elegance to all my written work including our published papers and this dissertation. Prof. Andrea's "Distributed Systems" course helped me develop a structured approach to comprehend complex distributed architectures and also encouraged me to further learn about systems. I took a special topics course on "Virtualization" from Prof. Remzi and thoroughly enjoyed it. I admired how he steered the class to understand the paper best by asking the hard questions. I was lucky to continue working on my class project as my research project which eventually got published and now forms a chapter in this

dissertation. I have benefited a lot by having the privilege of working closely with them and observing them over the years. They have helped me not just in technical aspects but have also taught me life lessons over the last several years. They have helped me overcome several hurdles that I have faced. Once advisors, always advisors. I am happy that I can always request them for advise. I will always strive to be a good student worthy of advisors like them.

I am also extremely thankful to Prof. Aditya Akella, Prof. Jonathan Eckhardt and Prof. Michael Swift for graciously agreeing to be part of my committee and for their valuable feedback and comments. Prof. Swift's "Operating Systems" course, which I took in my first semester, helped me learn the fundamentals and motivated me to pursue my research in this field. The numerous detailed questions and comments provided by Prof. Swift during my preliminary exam, my defense talk and my dissertation helped me improve the quality of my research output. I took courses on "Networking" and "Software Defined Networks" from Prof. Akella and I thoroughly enjoyed them and also learned a lot because there was always more emphasis on the learning outcome rather than traditional class procedures. I took Prof. Eckhardt's course on "Venture Creation" as a requirement for my minor but it instilled in me a desire to learn more about "Entrepreneurship" which led me to earn a Certificate in Entrepreneurship from the UW School of Business.

I am greatly indebted to the Computer Sciences department and the University of Wisconsin for accepting me into their prestigious graduate program and for providing me with all the necessary facilities during my entire graduate study. I also thank the many Professors whose courses helped me gain knowledge and always created a positive impact on my daily work life. I feel privileged to have learned these courses alongside exceptional students who were always a source of inspiration.

I am thankful to Nitin Agarwal for on-boarding me into his final Ph.D. research work and constantly helping me in implementing it which won a best paper award at the File

and Storage Technologies conference. I am also indebted to Nitin Agarwal, Cristian Ungureanu and many other esteemed researchers for mentoring me and providing a very pleasant experience during my internship at NEC Labs, Princeton. I also thank my several friends and seniors during my tenure at Amazon who helped me learn essential skills that were useful during my graduate studies and beyond.

I am fortunate to have had the opportunity to work with smart and hardworking colleagues from the Computer Sciences department who are also my friends: Nitin Agrawal, Ishani Ahuja, Ramanatthan Alagappan, Samer Al-Kiswany, Ashok Anand, Lakshmi Bairavasundaram, Siddharth Barman, Jayaram Bobba, Vijay Chidambaram, Thanh Do, Chris Dragga, Ramakrishnan Durairajan, Aishwarya Ganesan, Akhil Guliani, Haryadi Gunawi, Tyler Harter, Jun He, Kevin Houck, Joy James Prabhu, Sudarsun Kannan, Junaid Khalid, Rustam Lalkaka, Jing Liu, Lanyue Lu, Ao Ma, Joe Meehean, Piramanayagam Arumuga Nainar, Sanketh Nalli, Sriraam Natarajan, Ed Oakes, Sankaralingam Panneerselvam, Yuvraj Patel, James Paton, Thanumalayan Pillai, Abhishek Rajimwale, Deepak Ramamurthi, Matthew Renzelman, Mohit Saxena, Sayandeep Sen , Vivek Shrivastava, Srinath Sridharan, Sriram Subramaniam, Swaminathan Sundararaman, Venkatanathan Varadarajan, Laxman Visampalli, Raajay Viswanathan , Zev Weiss, Kan Wu, Leon Yang, Suli Yang, Jongwon Yoon , Wei Zhang, Yiying Zhang, Yupu Zhang and Dennis Zhou.

I thank my family for their continuous support, prayers to God, and their patience during the several years that I took to finish my graduate studies. They were my pillars of strength and my source of happiness during the many troubling times. They celebrated my small victories with great appreciation and happiness. My parents took great efforts to lay a very strong foundation in my childhood that has helped me survive the tempestuous times later in my adulthood. I am also indebted to my parents-in-law for their prayers to God, constant encouragement and faith in me. My wife Vinoliya Sebastian provided great support and comforting companionship that made

life easier. Although I often could not spend lots of time with her, she understood my time commitments and encouraged me in finishing my graduate studies. Her continuous encouragement and her prayers to God have had a tremendous impact on me. I thank my dear baby daughter Snowlina for supporting me in her own way by being a continuous source of joy and by not crying often. Her arrival in our lives has brought many blessings. My brother Joy James Prabhu helped me deviate a bit with his phone calls and video chats. His questions about both my work and life have had tremendous impacts on me. I am fortunate to have him as my brother. I also thank my other relatives who have helped motivate me when I was not in high spirits and have had a positive impact on my life.

# Contents

# List of Tables

# List of Figures

**NON-INVASIVE I/O CLASSIFICATION TECHNIQUES AND APPLICATIONS**

Leo Prasath Arulraj

Under the supervision of

Professors Andrea C. Arpaci-Dusseau and Remzi H. Arpaci-Dusseau

At the University of Wisconsin-Madison

Storage researchers and developers strive towards creating systems that store and retrieve data in correct, efficient, available, reliable, durable, fault-tolerant and cost-effective manner. Achieving these desirable properties for all I/O requests is a very hard challenge. I/O classification is an effective technique to mitigate this. I/O requests can be categorized based on their properties and better quality of service, performance and reliability be provided to the important I/O classes. This dissertation develops three novel non-invasive I/O classification techniques that work with many different file systems without significant additional implementation effort. Non-invasive techniques that do not require extensive modifications to existing systems face little resistance while making their way into current storage systems.

We build three diverse applications using our novel non-invasive I/O classification techniques. Our first application, called David, accurately emulates huge, and possibly futuristic, storage disks using a small physical disk with the aid of storage models. Since benchmark applications do not use actual I/O content from real applications, David uses our technique to identify and persistently store only the file system metadata I/O. Sky, our second application, is a hypervisor extension that improves the performance of caching and deduplication systems by gathering insights about guest I/O workloads using system-call interception. Smart caching gives higher priority to small files and file system metadata while smart deduplication differentially treats encrypted and file-copy I/O requests. Finally, we study the corruption resilience of the check-and-repair tools in three modern NoSQL distributed stores – Cassandra, MongoDB and Riak – and use

what we learned to build DSCK, a framework for improving their corruption resilience. We empirically show that these tools have poor corruption resilience. We then analyze the on-disk files of these systems and use what we learned to guide the design of DSCK. DSCK classifies I/O at file-granularity and transparently replicates and checksums only a selected set of files for corruption resilience. We use DSCK to build $\text{DSCK}_{\text{Cassandra}}$, a check-and-repair tool for Cassandra, that imposes low overheads but significantly improves corruption resilience.

# Abstract

Storage systems that can efficiently and reliably handle the vast amounts of data being generated today are the need of the hour. Data plays a crucial role in our daily lives. Storage developers and researchers need to constantly innovate to keep up with the challenge posed by the tremendous growth of data - it is expected that the size of all the data used by humans will grow to be 163 ZB by 2025 [157]. It is also equally important to get these innovations into real existing storage systems. Non-invasive innovations that do not extensively alter an existing storage system are easier to be adopted into existing storage systems because of the ease to deploy and maintain them. Moreover, non-invasive solutions that work with different configurations of a storage system without the need for significant additional effort are much more likely to become a part of existing storage systems.

I/O classification is an effective technique to tackle some of the problems posed by the vast amounts of data. I/O classification can be defined as the process of grouping I/O requests into different classes based on their properties and processing them differently in order to enhance various desirables like performance [22, 131, 187, 189, 208], reliability [73, 188], better security [186, 187, 189] and easier storage administration [73]. For example, a file system could classify I/O requests based on the destination file's properties in order to enable replication, encryption and compression only for certain files. I/O classification provides better service to only the important I/O classes.

This dissertation develops three new non-invasive I/O classification techniques

that work with many different file systems without requiring significant additional effort. Non-invasive techniques do not require pervasive changes across the storage stack and interfaces which makes them easier to adopt in existing storage systems. We then use these techniques to develop three novel applications in diverse areas such as benchmarking, virtual machines and distributed storage.

David, our first application, is an emulator that allows benchmarking futuristic large storage devices with existing small disks. The key observation used by David is that most benchmarking applications do not use the actual data from real applications but they stress the disk using an I/O pattern similar to the real application they represent. Therefore, David avoids storing the benchmark application's data on disk but instead just stores the file system metadata on disk in a space-efficient fashion. David therefore needs only a fraction of the actual disk space used by the benchmark application. David uses an accurate model of the emulated disk to respond to the I/O requests from the benchmark application after a sufficient delay corresponding to the amount of time the emulated disk would have needed to service that particular I/O request.

David heavily relies on accurately classifying application data from file system metadata. To achieve this, we develop a novel technique based on the observation that: all user data requests originate from the userspace through system calls but file system metadata I/O requests originate from the kernel. Our technique first captures the physical addresses of pages behind I/O related write system calls in a hash table. Subsequently, David, when handling an I/O request to an emulated disk, checks for the presence of an I/O request's backing page in the hash table to decide if the I/O request contains user data. David is able to accurately emulate large storage disks for many workloads with less than 3% error. We believe that the error rate can be further reduced with more accurate storage models of the emulated disk.

Sky, our second application, is a hypervisor extension that implements a smart caching and deduplication system using I/O classification insights that it gathers about

guest I/O applications. Sky uses the virtualization extension in modern processors to intercept the entry and exit of I/O related system calls in the guest VM and analyzes their arguments and return values to gather insights. Sky is able to classify I/O requests based on: the size of the associated files, whether they contain user data or file system metadata, are they likely to be encrypted and are they part of a file copy. The overheads imposed by Sky is small enough to improve the performance of a cache using an LRU policy for various workloads running on virtualized storage: a file name search workload runs 3.6 to 4.6 times faster, a TPC-H query runs 2.3 to 8.8 times faster, a file copy on a deduplicated storage is 5.5 to 8.3 times faster and finally, a file encryption on a deduplicated storage is 4.5 to 18.7 times faster.

DSCK, our third application, is a framework that helps implement corruption-resilient check and repair tools for distributed NoSQL storage systems. DSCK relies on classifying I/O requests based on the destination file's semantic information to facilitate implementing a better check and repair tool. DSCK uses a policy configuration file that specifies classification using file-path prefixes, patterns and suffixes. In building DSCK, we first study the corruption resilience of existing check and repair tools that come with three popular distributed storage systems - Cassandra, MongoDB and Riak - that significantly differ from each other in their design. Our study shows that the current tools for Cassandra and MongoDB have very poor corruption resilience. Riak has better corruption resilience but still has issues. We also study the corruption-resilience capability of these distributed systems guided by the following questions: how are files laid out on the disk?, what type of semantic information do the files contain?, how hard is to recover a file after corruption?, and does a file have checksums to aid detecting a corruption?

We design DSCK by using the observations and learnings from our study. These distributed systems have checksums to detect corruption to most user stored data but lack checksum protection for small critical system files. These critical system files are

also hard to recover after a corruption because they often need new specialized recovery tools to be built. Our study also showed that corrupted files can be recovered using different techniques.

The three components that make up DSCK are the corruption-resilient store, the checker and the repairer. The corruption-resilient store takes advantage of the fact that all these distributed systems are written in the userspace and use standard libraries like *libc* and *libaio* to interact with the storage layer. DSCK classifies I/O requests at the library-call level in order to transparently maintain local checksummed replicas for a selected set of files. Such I/O classification is non invasive because it does not need modifications to the NoSQL store or the file system. Additionally, it does not require super-user privileges and can work with different NoSQL stores and file systems. DSCK classifies files based on the semantic file type and employs file-type specific checking and recovery. We used DSCK to build a corruption resilient check and repair tool for Cassandra, called $\text{DSCK}_{\texttt{Cassandra}}$, that imposes negligible performance overhead but improves corruption resilience from 37.5% to nearly 100% of files stored by Cassandra; we also show that local node repair enables full-node restore in minutes rather than hours.

# Chapter 1

# Introduction

Digitally stored data has become an important aspect of almost all walks of human life over the past few decades. Correct, efficient, available, reliable, durable, fault-tolerant, cost-effective storage and retrieval of large quantities of data is a key building block for many of today's businesses like stock markets, banks, e-commerce, web servers, smart phones, cloud computing, movies, entertainment, photography, education, huge scientific experiments, high performance computing, space exploration, weather forecasting and health care. This is evidence that storage technology plays a critical role in our daily lives. Storage technology has come a long way over the past few decades. For example, the cost per GB of storage has decreased from approximately 10 million USD in 1956 to 6 cents in 2013 [191]. It is imperative for storage researchers and developers to understand the production, storage and consumption of digital data in the past and present to design the storage systems of the future.

Several storage technology trends pose a huge challenge for storage designers, developers and researchers to continuously innovate and build better storage systems. On the one end, with computers becoming more and more powerful, the need for faster storage devices and other techniques to hide the storage latency is increasingly important. Over the last 30 years, the processor cycle time has improved by over 2500

times [78]. However, the magnetic disk seek time improved only by 30 times. On the other end, the amount of data generated is increasing at a staggering rate - it is projected to increase from approximately 16 ZB in 2017 to 163 ZB by 2025 [157]. Additionally, on the hardware front, a variety of new storage technologies like Phase Change Memory [221, 231], Non Volatile Memory [223], Shingled Magnetic Disks [1, 2] are on the cusp of widespread adoption. These new storage devices that have different physical properties pose challenges in designing efficient software to manage the data stored in them. The storage disk capacities has been increasing at a staggering rate over the past few decades. This dissertation takes a step towards tackling these challenges by contributing three novel applications in diverse but important areas within the realms of storage. In the following paragraphs, we will pose three problems faced today in the world of storage and introduce a solution using our novel I/O classification techniques.

**Need for scalable benchmarking:** Developing efficient storage software (e.g. file system) is a very challenging task. Scientific measurements can help evaluate different design choices and choose the right ones. Benchmarking enables a storage researcher or developer to compare competing ideas through measurements and guides the research process in the right direction by pointing out the areas that need improvement. Benchmarking shapes a field [147]. Storage software of the future must manage data efficiently on a variety of storage disk technologies. Moreover, future hard disks will have significantly higher capacity than those available today. Hard disks are doubling in capacity approximately every 2 years [91, 121]. Designing storage software for the growing disk sizes is not trivial – there are interesting problems that show up with scale. For example, the flash translation layer in solid state drives that maps logical disk blocks to physical blocks poses interesting design tradeoffs between cost, performance and complexity. How can one benchmark software on the futuristic huge capacity disks when they are not physically available yet?

**A novel scalable benchmarking tool:** Simulation and emulation are widely used, proven techniques for researching future systems that are not yet available [30, 33, 164]. However, simulation can hide the benefits by not capturing complex "real system" effects and by not allowing real application workloads [71]. So, we combine the two themes of benchmarking and emulation to develop a novel tool that helps efficiently benchmark software on futuristic, huge-capacity storage disks that are not yet available. Our tool uses physical storage space that is only a fraction of the emulated disk space thereby facilitating scalable benchmarking. Moreover, our tool works with many different file systems without significant additional effort. Our tool has also been used to emulate a futuristic solid state drive with a novel interface called "nameless writes" that eliminates the need for huge memory requirements within the disk [229].

**Poor virtualized-storage performance and a solution:** Managing vast amounts of data is a tough task that requires lot of attention. Several commercial vendors [45] have taken advantage of this opportunity by allowing customers to rent storage for costs much cheaper than buying and managing them – this is made possible by the economy of scale. Virtualization and virtual machines are part of the key technology powering this market known as "cloud computing". Hypervisors or virtual machine monitors enable running multiple virtual machines on a single physical host. Virtual machines bring several advantages with them and one important advantage is improved resource utilization through sharing of resources among different users. There is however a performance problem with this approach – there are overheads due to virtualization and layering of software components. An effective technique to handle this is to reserve higher performance for the more important I/O – e.g. from a premium customer or for file system metadata that is essential to access data. However, it is very hard to perform such differentiation on I/O requests at the storage layer because of the lack of enough information. How can a hypervisor obtain information about I/O requests in a easily

deployable fashion without modifying storage layers and interfaces?

**Classifying I/O to improve performance:** The storage stack in the host operating system does not have any information to differentiate I/O from different customers. This is because of the standard interfaces between different layers of the storage stack that have slowly evolved over time. These interfaces are simple and robust but are also restrictive in allowing information flow to the lower levels of the storage stack. We develop a novel technique to bridge this semantic gap between layers of the virtualized storage stack using the hardware mechanism present in modern processors with virtualization extensions. We generate I/O classification insights at the hypervisor using technique and use the insights to enhance the virtualized storage performance by building smart caching and deduplication systems.

**Robust check and repair for modern NoSQL stores:** Ensuring the availability and durability of data amidst failures at the scale of terabytes and petabytes is a complex task. A research study of a large installation of approximately 1.5 million disks shows that data corruptions are not uncommon: upto 4% of drives exhibit data corruption in a period of 17 months [20]. It is very important for the storage software managing huge amounts of data to tolerate data corruptions. A specific kind of distributed storage systems called "NoSQL" stores are known for their ability to scale and store vast amounts of data. Such distributed NoSQL stores play an increasingly important role in internet-scale services and applications. For example, Apple and Netflix have thousand node installations of Cassandra NOSQL store [138]. NoSQL stores are often used in a configuration that replicates a single piece of information to three or more different nodes in isolated fault domains. The expectation is that such replication across different fault domains almost completely eliminates the possibility of data loss; any disk corruption can be repaired by using the other two replicas. However, we empirically show that these systems are not

completely resilient to corruption: they do not checksum or replicate the local metadata which is essential to access the data; they often do not use redundancy to recover from corruption; they often crash when encountering corruption [138]. We develop a framework for building corruption-resilient check and repair tools for the distributed systems. Our framework improves the corruption resilience and the corruption recovery time.

Researchers often face the challenge of innovating without the luxury of starting their design from scratch. Innovating under the constraints of the existing storage systems is more harder than developing novel solutions from scratch with complete freedom [13]. Innovations that pose fewer challenges to adopt, deploy and maintain alongside existing storage systems are more likely to attain success. We call such innovations "non-invasive" because they do not require extensive modifications across the storage stack. However, non-invasive innovations that work around existing aspects of a storage system, like its interface, often become complex and they need additional implementation effort to make them work with a different storage system. For example, they need significant changes to work with a different file system or a different storage technology like solid state drive, shingled drive etc. Non-invasive solutions that do not need additional efforts to make them work with a different configuration of the storage environment, which are the focus of this dissertation, have much higher chances of successful adoption in real storage deployments.

## 1.1 I/O Classification

Classifying I/O and treating them differentially is an effective technique to achieve desirable properties like better performance and quality-of-service in a storage system. While conventional storage systems treat all I/O requests similarly, I/O classification allows prioritizing and providing better service to an important subset of the I/O

requests. With the increasing usage of shared storage systems to lower the costs in infrastructure settings, I/O classification is essential to deliver on the SLAs (Service Level Agreements).

I/O classification can be defined as the process of segregating I/O requests into different classes based on their properties and processing them differently in order to enhance various desirables like performance [22, 131, 187, 189, 208], reliability [73, 188], better security [186, 187, 189] and easier storage administration [73]. For example, a storage application can classify I/O requests based on the user associated with it; an operating system can classify I/O requests based on the frequency of access to the associated content; a disk drive can classify I/O requests based on whether they contain user-supplied data or file system metadata. However, there are challenges associated with classifying I/O.

### 1.1.1 Challenges with I/O Classification and Past Solutions

Storage systems are made of several layers/components that talk to each other through well defined interfaces. Such modularity has several advantages like "easier development and testing", "interchangeable implementations of modules that offer different trade-offs" and "easier prototyping of changes" [213]. I/O requests take several forms as they pass through the various layers of the storage stack. Each layer of the storage talks to the other layer through a well-defined interface that has evolved over several decades to include only the information that is needed by the majority of the use cases.

Storage interface changes that could improve only a few usage scenarios often are lost over time and do not become part of the standard interface. Some storage interfaces can accommodate new changes when needed over time while others are almost impossible to change due to reasons like necessity to modify the hardware firmware, backward compatibility, approval from the governing standards committee and the associated

implementation complexity. For example, the interface between the *virtual file system (VFS) layer* in Linux operating system and the actual file systems can be changed when necessary with moderate difficulty by following a standard procedure of code testing and review by others in the kernel community [108]. However, disk interfaces like ATA and SCSI [10] are much harder to change because they are governed by a standards committee that accommodates changes after a more stringent process that involves voting, thorough discussions and debate [202]. Another huge motivation to keep interfaces simple is to avoid complexity that eventually leads to implementations that are not robust and reliable. These historical trends have made storage interfaces very restrictive over time. The lower a layer is in the storage stack, the lesser the amount of information it has about the I/O request it is processing.

Storage systems are for the most part forced to treat all I/O requests similarly due to the traditionally restrictive and simple interfaces they provide. Expectations on performance and reliability for the data stored cannot be conveyed to the storage system. This is problematic because information about I/O requests is essential for making decisions that will lead to improved reliability, performance and cost benefits. This loss of information is termed as "Semantic Gap" in the storage literature and bridging this semantic gap has been actively researched [13, 22, 131, 187, 189].

One obvious solution is to make the storage interface more expressive so that the file systems can communicate more details to the storage devices [73, 129, 131, 185, 225]. However, a major hurdle to such explicit I/O classification is the required modifications across all layers and connecting interfaces of the storage stack for both generating and passing down classifications. Modification to all component layers is not feasible in many scenarios. Sometimes, the software components are not open source and cannot be changed. And in some other scenarios, the work required to implement and maintain information-passing interfaces is too high. If the interfaces are governed by standards [131] or are implemented in hardware, it is hard to get them changed. Any

new approach that requires wide-scale top-to-bottom software modification will likely be challenging to maintain and deploy for existing systems.

Other solutions advocate making the file systems learn about the internals of the storage disks and taking advantage of it or vice versa. Many solutions under this category are non-invasive because they do not involve changes to the interfaces between the storage components. Such solutions are easier to adopt, deploy and maintain when compared to those solutions that require modifying the interfaces. However, these solutions also have some disadvantages. Because such non-invasive solutions often tend to infer properties of different subcomponents of the storage system, they are embedded with the graybox [12] information about these subcomponents. This necessitates additional implementation effort to make such non-invasive solutions work with a different choice of storage subcomponents like the file system or the storage disk. Moreover, storage devices are rapidly evolving with a plethora of internal technologies, like Non-volatile Memory [223], Shingled Magnetic Disks [1, 2] and Phase Change Memory [221, 231], making it challenging to make the file systems adapt to all of them. Another impediment to such solutions is the fact that many details about the internal implementations within disks are not revealed publicly by the disk manufacturers [7, 203].

Imparting knowledge and intelligence about file systems to the storage disks has its own disadvantages. The storage disks have to learn about the on-disk data structures laid out by a variety of file systems and optimize for that. This adds more complexity to the disk firmware leading to robustness concerns. Moreover, the on-disk structures of a file system occasionally change over time and keeping the firmware updated is a cumbersome task. A bigger hurdle is posed by the less powerful memory and compute resources within the disk that need to be upgraded to handle the additional complexity thereby increasing the total disk manufacturing costs. Many hardware vendors are therefore averse to adding more complexity to the firmware.

### 1.1.2 Non-Invasive I/O Classification

In order to avoid the disadvantages with these earlier approaches, we achieve I/O classification using new techniques that neither change the interfaces nor require extensive modifications to support different configurations of the storage system with alternate subcomponents. Specifically, we discuss three non-invasive I/O classification techniques in this dissertation that also work with a variety of file systems without any additional implementation effort. We also use these techniques to build and evaluate novel applications in diverse areas like benchmarking, caching, deduplication and distributed system check and repair. Based on our evaluation, these techniques impose tolerable overheads making them usable in diverse scenarios like: a "scale-down" benchmarking emulator, an intelligent caching and deduplication system for virtualized storage and a corruption resilient check and repair tool for distributed storage systems. The following Sections 1.2 to 1.4 give a brief overview of the three techniques and their applications saving the details for the subsequent Chapters 2 to 4 of this dissertation.

## 1.2 David: Emulating Goliath Storage Devices

Benchmarking the performance of various applications and workloads on different kinds of storage devices is essential in making the right design choices when developing a storage application or a future storage device. David is a novel benchmarking tool that allows emulating large storage devices using much smaller physical storage space with the aid of an inline performance model of the large/futuristic storage device. Using David, a storage developer or researcher can benchmark applications on a variety of storage devices without having to spend huge amounts of money and effort to procure them beforehand. David can also be used to benchmark futuristic storage devices that are not yet available in the market. For example, we successfully used David to benchmark

a new solid state drive (SSD) disk that introduces a novel interface called "nameless writes" [229].

The key observation enabling emulating large storage devices using much smaller physical storage is that most benchmark applications issue I/O requests to stress the storage device as the real application does but they do not use the actual content from the real application. For example, a mailserver benchmark [103] or a fileserver benchmark [124] does not use real mail messages or file contents but only issues I/O requests that mimic the pattern of a real mailserver or fileserver application respectively. David uses this insight in order to not store the data writes in physical storage but only stores the file system generated metadata after remapping them to a different destination location on the available physical disk.

David needs to correctly classify I/O requests containing file system metadata from the data generated by the benchmark application. We use two different non-invasive I/O classification techniques. The first technique, derived from past research work on Semantically Smart Storage [3, 13], uses knowledge about the file system and interposes on the I/O requests to the disk in order to parse and interpret their content to achieve block classification. Though this technique is non-invasive and easy to adopt and deploy, it needs additional implementation effort to make it work with a new file system. This is because the details about the new file system are necessary to parse and interpret the I/O requests from that file system to perform the I/O classification.

To alleviate this, we develop a new non-invasive I/O classification technique that works with many file systems without any additional implementation effort. The key observation behind this new technique is that all data I/O requests originate from the userspace benchmark application through system calls while the file system metadata originates from the file system within the kernel. Therefore, we modify parts of the operating system in order to add the physical memory addresses of the pages behind the I/O related system calls into a hash table. Subsequently, when David intercepts

and processes the I/O requests to the emulated disk, it checks if the pages behind the I/O request are present in the hash table. Presence indicates that it is a benchmark application issued data I/O request while absence indicates that it is a file system generated metadata I/O request. This technique captures page addresses above the file system and hence is not affected by the choice of a different file system. However, if the file system does special processing like encryption or compression that move the data into a new page, then some more implementation effort is required to capture the addresses of the final destination pages.

We evaluated the above two techniques and found that they yield similar I/O classification accuracy but the second technique has lesser overheads when compared to the first technique. It is also less complex, easier to implement and works with most file systems without additional special effort. We evaluated our new technique by using David with the Btrfs file system. David, when using our new technique, was able to predict the runtime of many workloads including "file search" and "fileserver, mailserver and webserver benchmarks" with under 3% error. We believe that David will be even more accurate with a more accurate disk drive model. Our current storage model for the magnetic disk drive does not model the on-disk cache accurately.

## 1.3 Sky: Improving Virtualized Storage Performance

Virtualization has become ubiquitous in data center environments and serves as a key technology enabling cloud computing. Virtualized storage provides several advantages like lower cost of ownership, easy snapshots/backups, and easier storage administration. However, the performance of virtualized storage is not as good as native storage devices due to several reasons including "virtualization costs" and "long I/O path traversing several layers". The I/O path from the application to the physical disk is often complex and composed of several layers spanning multiple servers in such virtualized storage

systems [208].

A promising approach to improve performance through additional information is to *explicitly* classify I/O requests into different classes and treat these classes differentially [122, 131, 208]. Better performance, quality-of-service, availability, reliability, durability and security can be reserved for the important I/O classes, for example: by caching them with higher priority, granting more network bandwidth to them, storing them on a high performance storage media, creating more replicas, scrubbing them frequently, encrypting them or skipping deduplication [25, 122, 189, 199, 214]. The lower layers in such multi-layered storage stacks do not have complete information about what the upper layers are doing due to the simple and restrictive interfaces governed by standards. As a result, lower layers are information impoverished, and cannot implement performance optimizations that require such high-level knowledge.

We implement an improved caching system for virtualized storage that gives higher priority to small files and file system metadata. We also implement a smart deduplication system that avoids deduplicating encrypted content and caches the mapping information for file-copy content that will be written back again soon. Building such smart caching and deduplication system heavily relies on classifying I/O requests based on size, metadata vs. data, encrypted writes, file copy I/O pattern.

Changing all the interfaces across the storage layers to enable I/O classification is hard to adopt, deploy and maintain because of the pervasive changes it requires across the storage stack. Non-invasive graybox [12] techniques, that use details about the storage components higher up in the storage stack in order to infer the required classification information, help overcome this issue but they need additional implementation effort in order to support a different storage component, like a new file system.

To overcome these issues, we develop a novel non-invasive I/O classification technique for virtualized storage that works for most file systems without requiring additional implementation effort. We then build a hypervisor extension called Sky, in the

KVM [109] virtual machine monitor, that implements smart caching and deduplication. Sky [16] uses the hardware virtualization extensions present in modern processors to intercept system call entry and exit of I/O related guest applications from within the hypervisor. Sky performs additional processing on the arguments to the I/O related system calls and their return values to gather insights for classifying I/O requests. Chapter 3 details this technique explaining how the insights are gathered and used to build a smart caching and deduplication system.

Sky achieves similar I/O classification insights as past research work that needed modifications to the interfaces [131, 206]. Moreover, the overheads imposed by our I/O classification technique are modest enough to use them in building applications like smart caching and deduplication system. We show the benefits of Sky by using three case studies.

In the first case study, Sky is used to gather information about guest I/O like the block-lifetimes from the hypervisor. The second case study is about a smart cache in the hypervisor that prioritizes file system metadata and small files over large files. Our smart cache improves the performance of file-name search workload and database application by 3.6 to 4.6 times and 2.3 to 8.8 times respectively. In our final case study, we use Sky to improve the performance of a deduplication system in hypervisor by identifying encryption and file copy I/O patterns. For encryption workloads, Sky skips deduplication because encrypted content is known to be mostly unique and a bad candidate for deduplication [206]. For file copy I/O pattern, Sky caches the mapping between the payload checksum to the deduplicated disk location during reads so that during the subsequent write of the same content, expensive disk backed mapping table lookups can be avoided using this cache. We also show that our novel classification technique that intercepts system calls is portable across operating systems- Sky supports both Linux and FreeBSD guest operating systems. Such portability is possible because the system call interface is similar across modern operating systems due to POSIX

standards.

## 1.4 Corruption Resilient Check and Repair

Distributed NoSQL stores are used by many organizations [83, 86, 89, 170] for big-data analytics and real-time web applications. Ever increasing installation-sizes of these distributed storage systems, comprising of hundreds of machines and thousands of disks [138], makes storage component failures the norm rather than the exception [66, 174]. Data corruptions can occur due to a variety of reasons including drive failures [10, 14, 23, 48, 56, 66, 70, 92, 102, 112, 133, 140, 146, 153, 156, 174, 175, 178, 179, 204], RAM failures [35, 176, 194] and bugs in software [20, 26, 43, 61, 156, 173, 177, 201, 226]. Anecdotal evidence has shown the prevalence of storage errors and corruptions [136, 139, 165]. Data corruptions affect data stored in frequently accessed disks as well as snapshots and backups in archival storage [18, 20, 77, 119, 134, 144, 173, 177]. Corruption-free snapshots are essential for recovery from failures, analytics, operations and other applications like *Back in Time Execution (BITE)* [168, 182] that allow executing queries over historical snapshots of a database for business reasons including auditing [104, 181, 183, 197, 211].

Failures are a fact that must be coped with, not problems to be solved [148]. Disk failures, if not coped with in a timely fashion, can lead to significant losses. Each hour of downtime can be costly, from $200,000 per hour for an Internet service like Amazon to $6,000,000 per hour for a stock brokerage firm [107, 148]. Downtime and data loss combine to cost companies and end-users billions of dollars each year [106, 190].

Consequently, a lot of importance is being given to making systems robust to failures and highly available. Jim Gray has called for *Trouble-Free Systems* that are used by millions of people every day and yet managed by a single part-time person [69]. Patterson et al. hypothesize that the recovery performance is more fruitful for the research community and more important for society than traditional performance in the 21st century [148].

The authors suggest *Recovery Oriented Computing* and stress the importance of fast repair because it improves dependability and lowers the cost of ownership. Unfortunately, we empirically find that the check and repair tools that come with modern distributed NoSQL stores like MongoDB and Cassandra are not corruption-resilient (details in Subsection 4.1.1).

A straightforward technique to handle data corruptions in modern distributed storage systems with replicas is to take the corrupted node out of the cluster, replace it with a new node and bootstrap it with data from the other live replicas. However, this technique has a few disadvantages: it consumes several hours of time, it consumes significant network bandwidth for transferring the data from the live replicas to the new node, it falls short of the user's reliability expectations because of the loss of a replica until the recovery is completed and most importantly, this technique cannot be used to recover corrupted disk-snapshots because there are no online replicas to fetch the lost data from. There is a need for a fast corruption-resilient check and repair tool that brings benefits like improved dependability and lower cost of ownership. The need for good check and repair tools is also evident from a recent research that finds that many modern distributed NoSQL stores do not consistently use redundancy to recover from file system faults - a single file-system fault can cause catastrophic outcomes such as data loss, corruption, and unavailability [62].

We perform a thorough study of the corruption-resilience capability of three distributed key-value stores: MongoDB [42], Cassandra [113] and Riak [110]. Our study is guided by the following questions: how does the NoSQL store lay out files on disk?, what type of semantic information do various files contain?, how hard is it to recover a file once it is corrupted?, does the file have checksums in order to detect a corruption? We empirically find that the check and repair tools that come with these distributed storage systems have poor corruption resilience. We also manually looked at the source code of the distributed storage systems and inspected the files they store on disk. We

conclude our study with a list of observations regarding the corruption resilience of these distributed storage systems and how they influenced the design of DSCK. One key observation from our study is that these NoSQL distributed storage systems protect most of the user-supplied data with checksums and only small critical system files, that are essential to access the user-supplied data, are not protected with checksums.

We use the results from our study to design and build a framework called DSCK that can be used to build corruption-resilient check and repair tools. DSCK uses the results from the study to classify files that need additional help in improving their corruption recoverability from other files. Such file classification is based on answers to questions like: does the file have checksums to detect a corruption? and how hard is it to recover the file after a corruption?. DSCK then intercepts and classifies the I/O related library calls made by the distributed system in order to keep transparent checksummed local replicas of only a selected set of files. Files that do not have checksum protection from the distributed system or file that do not have existing tools to recover them after a corruption can benefit from such local checksummed replication. DSCK uses a configuration file specific to the distributed system in order to classify the I/O requests at the library-call level. The configuration file uses file-path prefixes, patterns and suffixes to specify the file classification.

Intercepting the library calls makes our implementation non-invasive because it can work with many different distributed storage systems that run on a variety of local file systems without any additional implementation effort. Moreover, our interception does not require super-user or administrator privileges. One alternate invasive approach would have been to modify the NoSQL store, the lower level storage system and the system call interface to explicitly specify which files need replicas and checksums. Another alternate approach is to just modify the NoSQL store to add checksums to files that don't have checksums. However, both these invasive approaches require modifications to every NoSQL store that needs to be supported. Also, maintaining

backwards compatibility with previous release versions of the NoSQL store is a challenge. This makes them hard to adopt, deploy and maintain in existing real systems.

DSCK also uses I/O classification at the file granularity for checking and recovering a corrupted file by allowing pluggable scripts that handle a specific type of file. We use DSCK to implement $\text{DSCK}_{\text{Cassandra}}$, a corruption-resilient check and repair tool for Cassandra. We show through experiments that $\text{DSCK}_{\text{Cassandra}}$ imposes negligible performance overhead in the common case but improves corruption resilience from 37.5% to nearly 100% of files stored by Cassandra. We also show that local node repair enables full-node restore in minutes rather than hours.

The rest of this dissertation is organized as follows: in Chapter 2, we discuss the design, implementation and evaluation of David; Chapter 3 contains details about Sky; The results of our study and evaluation of the corruption resilience of modern distributed storage along with the details about DSCK framework is in Chapter 4. We discuss related work in Chapter 5 and conclude in Chapter 6. Chapter 6 also contains some of the learnings from our experience in building these systems and a list of future work.

# Chapter 2

# Emulating Goliath Storage Devices

For better or for worse, benchmarking shapes a field [147]. Benchmarking plays a very important role in guiding the design of future file and storage technology. In this chapter, we develop a novel non-invasive and easy to adopt I/O classification technique and use it to build a storage emulator, called David [5]. David allows storage developers and researchers to run benchmarks on a variety of devices, including futuristic large devices that are not yet physically available, by using accurate storage performance models. Today, there are a variety of new storage devices [1, 2, 38, 75, 93, 98, 132, 193] including Phase Change Memory [221, 231], Non Volatile Memory [223] and Shingled Magnetic Disks [1, 2] that vary significantly in the underlying technology. It is important for an application developer to run a benchmark on different types of storage disks to understand its behavior so that the application can be designed correctly to match the expectations when run on a variety of storage technologies. It is also extremely useful for a storage system developer or researcher to run various benchmarks with realistic configurations [135, 158, 192, 196, 209] on a storage technology that is under development so that the results can be used to guide the design of the storage disk. David is useful for performing both these kinds of analyses while requiring only a fraction of the storage space used by the application given an accurate performance

model for the storage disk is available. Such analyses help in identifying performance bottlenecks, fine-tuning optimizations, and for making design decisions [135].

The key observation on which David depends is that many benchmark applications like postmark [103] and filebench [124] do not use real file contents from the application they represent but use artificial content that they create. These benchmarks instead focus on replaying the same I/O pattern as the application that they represent. Therefore, it is sufficient and also beneficial for a storage emulator built for benchmarking purposes to not store the contents of individual files on physical disk, but only the structure and properties of the metadata. Since file data constitutes a significant fraction of the total file system size, ranging anywhere from 90 to 99% depending on the actual file-system image [6], avoiding the need to store file data has the potential to significantly reduce the required storage capacity during benchmarking.

David maintains a "compressed" version of the original file-system image for the purposes of benchmarking in which unneeded user data blocks (file contents) are omitted using novel I/O classification techniques to distinguish data from metadata at scale; file system metadata blocks (e.g., inodes, directories and indirect blocks) are stored compactly on the available backing store. This enables David to run various workloads at scale using backing storage disks that are only a fraction of the emulated disk size. David synthetically generates file contents when necessary to ensure that applications remain unaware of this interposition.

Classifying file system metadata from user stored data accurately at scale is critical for David. David employs two different techniques to achieve this classification that offer different tradeoffs. The first technique, which we call as "implicit classification", interposes on the I/O requests sent to the disk and uses file system knowledge in order to classify file system metadata from user data. This does not require any modifications to the file system or the operating system. Therefore, it is easy to adopt and run. This technique is derived from past research work on "Semantically Smart Disks". We

enhance this known technique to accurately classify I/O requests in a timely fashion at scale using a technique called "journal snooping". The second technique that David uses, which we call as "explicit classification", makes modifications to the system call layer of the operating system in order to classify file system metadata from data. Because almost all file systems are implemented inside the operating system kernel, only user-supplied data passes through system calls. File System metadata is generated within the kernel file system and hence it will not be intercepted at the system call layer. So, we capture the physical memory addresses of the page buffers that are used with I/O related system calls and use them while interposing I/O requests to the emulated disk to classify file system metadata from data. This technique is file system agnostic and therefore works with all file systems without any additional effort. In contrast,the first technique needs to be re-implemented for each new file system using the file system knowledge. David works under any file system; We implement and evaluate the first technique for the Ext3 file system [212]. We test that the second technique works with Btrfs [159], a log-structured file system that is very different from Ext3 in design.

Another important component of David is the storage model for the emulated disk drive. Since David modifies the original I/O patterns by not storing the user-supplied data on disk, it needs to model the runtime of the benchmark workload on the original uncompressed image. David uses an in-kernel model of the disk and storage stack to predict the run times of all individual requests as they would have executed on the uncompressed image. Our storage models model the physical magnetic disk and the I/O request queues with sufficient accuracy. Our in-kernel storage model is fairly accurate in spite of operating in real-time within the kernel, and for most workloads predicts a runtime within 5% of the actual runtime. For example, for the Filebench webserver workload, David provides a 1000-fold reduction in required storage capacity and predicts a runtime within 0.08% of the actual.

We briefly demonstrate that David is able to emulate multi-disk systems like RAID

using a small amount of memory. David maintains modeling state for each individual disk in the RAID array in order to accurately emulate the entire RAID array. Our implementation emulates a software RAID array. David can emulate a more complex, dedicated hardware RAID array provided we have a storage model for the hardware RAID device.

To demonstrate that David can be used to emulate futuristic storage disks, we use David to emulate a special large Solid State Drive (SSD) device that exports a new interface called "nameless writes" [229]. The *nameless writes* interface allows the storage stack (file system) to write data to the SSD without specifying the logical block address inside the disk. Instead, the disk chooses the destination physical location and reports it back to the file system. The file system stores this address in its metadata structures and uses it to fetch the block during future reads. This avoids the need for huge indirection tables within the SSD that map logical block addresses to the physical locations. Additionally, the complete flexibility to choose the on-disk physical location allows more efficient block allocation, garbage collection and wear leveling.

We additionally use David to emulate a 1 TB magnetic disk while using a 80 GB disk as the physical backing store. This allowed us to validate that David is able to accurately predict benchmark times when emulating large disks.

The rest of this chapter is structured as follows. We first describe David's design in Section 2.1. We then discuss block classification, which forms a critical component of David, in Section 2.2). We detail "implicit block classification" which is derived from prior work and "explicit block classification" which is our new contribution in this dissertation. David uses block classification to skip application data writes and only layout the metadata blocks on disk after remapping them as explained in Section 2.3. We evaluate David in Section 2.4 and summarize in Section 2.5.

## 2.1 David's Design

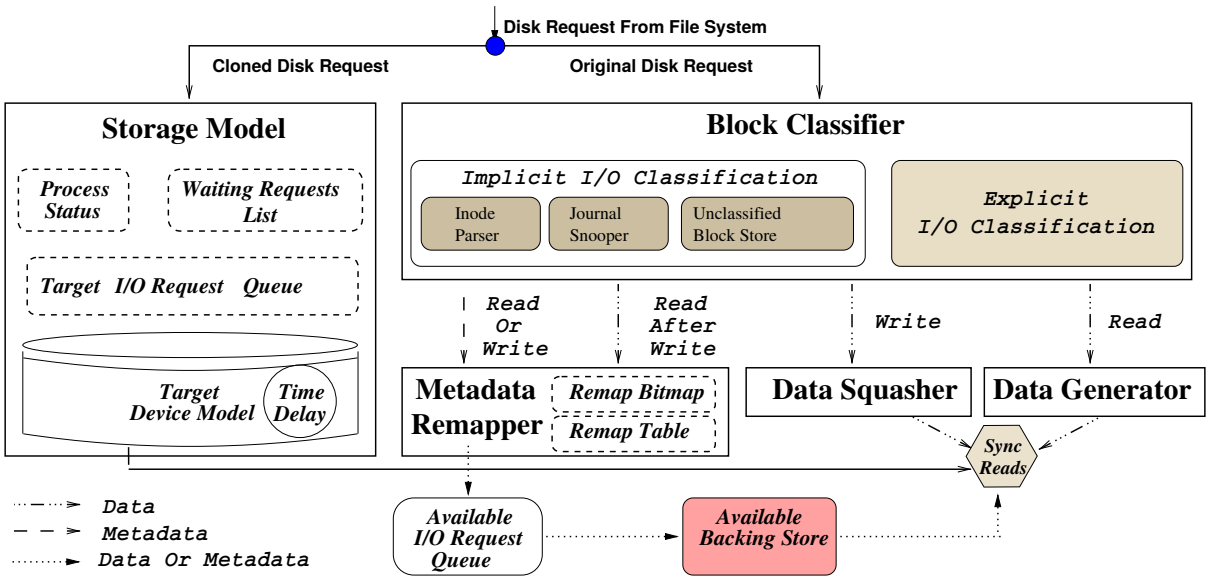



Figure 2.1: **David Architecture.** *Shows the components of David and the flow of requests handled within.*

This section presents the design and implementation of David by describing its constituent subsystems and its overall architecture.

David exports a fake storage stack including a fake device of a much higher capacity than available. This chapter uses the term *target* to denote the hypothetical larger storage device, and *available* to denote the physically available system on which David is running.

David is implemented as a pseudo-device driver that is situated below the file system and above the backing store, interposing on all I/O requests. This pseudo-device appears as a regular disk to the file system. Since the driver appears as a regular device, a file system can be created and mounted on it. Being a loadable module, David can be used without any change to the application, file system or the kernel. This non-invasive property makes David easy to adopt and run. Figure 2.1 presents the architecture of David with all the significant components and also shows the different types of requests that are handled within. The four important components of David are briefly described below and are discussed in detail later in Section 2.2 through Section 2.3.

**Block Classifier:** The Block Classifier is responsible for classifying blocks addressed in an I/O request as user-supplied data or file system metadata. This classification is used by David to prevent the I/O requests to data blocks from going to the backing store in order to save on the storage space necessary for the emulation. David intercepts all writes to data blocks, records the block addresses if necessary, and discards the actual write using the Data Squasher. I/O requests to metadata blocks are passed on to the *Metadata Remapper*.

**Metadata Remapper:** The Metadata Remapper is responsible for laying out metadata blocks more efficiently on the backing store. It intercepts all write requests to metadata blocks, generates a remapping for the set of blocks addressed, and writes out the metadata blocks to the remapped locations. The remapping is stored in the Metadata Remapper to service subsequent reads to these metadata blocks.

**Data Squasher:** The Data Squasher squashes the data writes preventing them from being stored in the available disk. Such squashing does not lead to correctness issues for benchmarking applications.

**Data Generator:** Writes to data blocks are not saved by David, but reads to these blocks could still be issued by the benchmark application; in order to allow the benchmark applications to run transparently, the Data Generator is responsible for generating synthetic content to service subsequent reads to data blocks that were written earlier and discarded. The Data Generator contains a number of built-in schemes to generate different kinds of content and also allows the application to provide hints to generate more tailored content (e.g., binary files). Data Generator can be configured to just generate junk data for benchmark applications like filebench that do not care about the returned file contents.

**Storage Model:** David modifies the original I/O request stream because it skips storing writes to data blocks and remaps writes to metadata blocks. These modifications in the I/O traffic substantially change the application runtime rendering it useless for benchmarking. The Storage Model carefully models the target storage subsystem underneath to predict the benchmark runtime on the target system. By doing so in an online fashion with little overhead, the Storage Model makes it feasible to run large workloads in a space-efficient manner.

### 2.1.1 Choice of Available Backing Store

David is largely agnostic to the choice of the backing store for available storage: HDDs, SSDs, or memory can be used depending on the performance and capacity requirements of the target device being emulated. Through a significant reduction in the number of device I/Os, David compensates for its internal book-keeping overhead and also for small mismatches between the emulated and available device. However, if one wishes to emulate a device much faster than the available device, using random access memory is the ideal option. For example, as shown in Section 2.4.4, David successfully emulates a RAID-1 configuration using a limited amount of memory. We have also show that a fast SSD device can be emulated using David [229]. If the performance mismatch is not significant, a hard disk as backing store provides much greater scale in terms of storage capacity. Throughout this chapter, "available storage" refers to the backing store in a generic sense.

## 2.2 Block Classification

The primary requirement for David to skip data writes using the Data Squasher is the ability to classify a block as user-supplied data or file system metadata. David provides two types of block classification techniques called "implicit" and "explicit". The

implicit approach is more laborious but provides a flexible approach to run unmodified applications and file systems. The explicit I/O classification approach is straightforward and much simpler to implement, albeit at the cost of a small modification in the operating system; both are available in David and can be chosen according to the requirements of the evaluator. The implicit approach is demonstrated using Ext3 file system and the explicit approach using Btrfs file system.

### 2.2.1 Implicit I/O Classification

Implicit I/O classification is based on prior work on Semantically-Smart Disk Systems (SDS) [189]; an SDS employs three techniques to classify blocks: *direct* and *indirect* classification, and *association*. With direct classification, blocks are identified simply by their location on disk. Many block types in Ext2 and Ext3 file systems can be classified by using their location on disk because they are statically assigned for a given file system size and configuration at the time of file system creation. Some examples of these block types include the super block, the group descriptors, the inode and data bitmaps, the inode blocks and the blocks belonging to the file system journal. Moreover, the location of such blocks do not change during the lifetime of the file system. With indirect classification, blocks are identified only with additional information; for example, to identify directory data or indirect blocks, the corresponding inode must also be examined. Blocks that are dynamically-allocated need to be classified using indirect classification. Directory blocks, indirect (single, double, or triple indirect) blocks and data blocks are some examples. Finally, with association, a data block and its inode are connected.

There are two significant additional challenges David must address. First, as opposed to SDS, David has to ensure that no metadata blocks are ever misclassified. Second, benchmark scalability introduces additional memory pressure to handle scenarios where the blocks cannot be immediately classified as data or metadata but can only be classified

after a delay when other related blocks are also written to by the file system. In this dissertation, only new contributions to make implicit block classification work at scale are discussed and the details of the basic block-classification techniques can be found in the original SDS paper [189].

**Unclassified Block Store**

To infer when a file or directory is allocated and deallocated, David tracks writes to inode blocks, inode bitmaps and data bitmaps; to enumerate the indirect and directory blocks that belong to a particular file or directory, it uses the contents of the inode. It is often the case that the blocks pointed to by an inode are written out before the corresponding inode block; if a classification attempt is made when a block is being written, an indirect or directory block will be misclassified as an ordinary data block. This transient error is unacceptable for David since it leads to the "metadata" block being discarded prematurely and could cause irreparable damage to the file system. For example, if a directory or indirect block is accidentally discarded, it could lead to file system corruption.

To rectify this problem, David temporarily buffers in memory writes to all blocks which are not yet classified, inside the *Unclassified Block Store* (UBS). These write requests remain in the UBS until a classification is made possible upon the subsequent write of the corresponding inode. When a corresponding inode does get written, blocks that are classified as metadata are passed on to the Metadata Remapper for remapping; they are then written out to persistent storage at the remapped location. Blocks classified as data are discarded at that time. All entries in the UBS corresponding to that inode are also removed.

The UBS is implemented as a list of block I/O (*bio*) request structures. An extra reference to the memory pages pointed to by these bio structures is held by David as long as they remain in the UBS; this reference ensures that these pages are not mistakenly

freed until the UBS is able to classify and persist them on disk, if needed. The caching of unclassified by the UBS without persisting them to disk makes David not guarantee crash consistency. This is not a problem for benchmarking applications that David targets.

**Journal Snooping**

Storing unclassified blocks in the UBS can cause a strain on available memory in certain situations. In particular, when Ext3 is mounted on top of David in ordered journaling mode, all the data blocks are written to disk at journal-commit time but the metadata blocks are written to disk only at the checkpoint time which occurs much less frequently. This results in a temporary yet precarious build up of data blocks in the UBS even though they are bound to be squashed as soon as the corresponding inode is written; this situation is especially true when large files (e.g., 10s of GB) are written. Such additional memory pressure also triggers the checkpoint process in Ext3 so that it happens earlier than when it would have normally happened. In order to ensure the overall scalability of David, handling large files and the consequent explosion in memory consumption is critical. To achieve this without any modification to the Ext3 filesystem, David performs Journal Snooping.

David snoops on the journal commit traffic for inodes and indirect blocks logged within a committed transaction; this enables block classification even prior to checkpoint. When a journal-descriptor block is written as part of a transaction, David records the blocks that are being logged within that particular transaction. In addition, all journal writes within that transaction are cached in memory until the transaction is committed. After that, the inodes and their corresponding direct and indirect blocks are processed to allow block classification; the identified data blocks are squashed from the UBS and the identified metadata blocks are remapped and stored persistently. The challenge in implementing Journal Snooping was to handle the continuous stream of unordered

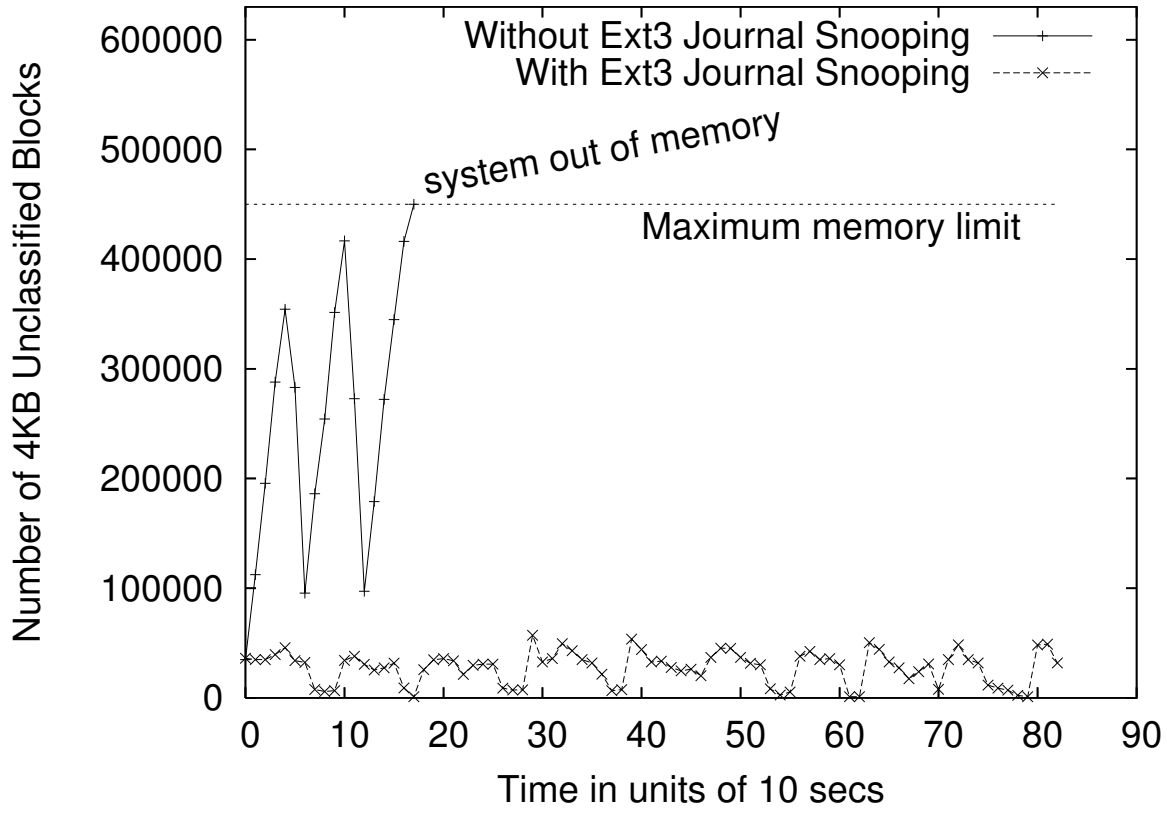



Figure 2.2: **Memory usage with Journal Snooping.** *This graph shows the number of 4 KB block requests present in the UBS sampled at 10 sec intervals.*

journal blocks and reconstruct the journal transaction.

Figure 2.2 compares the memory pressure with and without Journal Snooping demonstrating its effectiveness. It shows the number of 4 KB block I/O requests resident in the UBS sampled at 10 sec intervals during the creation of a 24 GB file on Ext3 ; the file system is mounted on top of David in ordered journaling mode with a commit interval of 5 secs. This experiment was run on a dual core machine with 2 GB memory. Since this workload is data write intensive, without Journal Snooping, the system runs out of memory when around 450,000 block I/O requests are in the UBS (occupying roughly 1.8 GB of memory). Journal Snooping ensures that the memory consumed by outstanding block I/O requests does not go beyond a maximum of 240 MB for this workload.

### 2.2.2 Explicit I/O Classification

David is meant to be useful for a wide variety of file systems; explicit I/O classification provides a mechanism to rapidly adopt a file system for use with David. Since data writes can come only from the benchmark application in user-space whereas metadata writes are issued by the file system, our approach is to identify the data blocks before they are even written to the file system. Our implementation of explicit I/O classification is thus file-system agnostic – it relies on a small modification to the system call layer of the operating system in order to collect additional information. We demonstrate the benefits of this approach using Btrfs, a file system quite unlike Ext3 in design.

While handling a write related system call, the operating system usually copies the data from the user buffer, passed as a system call argument, into a kernel allocated page in the page cache. David captures the physical addresses of the in-memory pages where the such data content is stored in a hash table. Subsequently, the operating system issues I/O requests corresponding to these system calls. David intercepts these writes when they reach the emulated disk. David then checks if the pages behind the I/O request are in the hash table populated earlier in order to decide whether the write is to file system metadata or application data. A presence in the hash table indicates that the I/O request contains benchmark application data while an absence indicates that the I/O request contains file system created metadata. Once the presence is tested, the pointer is removed from the hash table since the same page can be reused for metadata writes in the future. An alternate implementation could avoid the need for a hashtable by modifying the structure associated with buffer cache pages to store information about whether they contain benchmark application data.

There are certainly other ways to implement explicit I/O classification. One such alternate technique is to capture the checksum of the contents of the in-memory pages instead of their memory addresses in order to track data blocks. Another alternative

approach could modify the file system to explicitly flag the metadata blocks. We believe our approach is easier to implement, does not require any file system modification, and is also easier to extend to software RAID since parity blocks are automatically classified as metadata and not discarded. David's explicit I/O classification technique will need additional implementation effort to handle file systems that encrypt or compress data.

## 2.3 Metadata Remapping

Since David exports a target pseudo device of much higher capacity to the file system than the available storage device, the block I/O (bio) requests issued to the pseudo device will have addresses in the full target range and thus need to be suitably remapped. For this purpose, David maintains a remap table called Metadata Remapper which maps "target" addresses to "available" addresses. Although the Metadata Remapper has many similarities to other block re-mappers like the Flash Translation Layer in a SSD or the copy-on-write virtual-disk manager in a hypervisor, it differs from them in that it does not handle crash recovery. This is not a real problem because David targets benchmarking applications which can be rerun again in the rare case of a crash. David contains two implementations of the remap table: one uses a hash table while the other uses an interval tree. The hash table implementation allows fast lookups and inserts when using file systems like Ext3 that issue small write requests that always affect a fixed number of logical blocks. The interval tree implementation is more efficient when using a log structured file system like Btrfs that issues large write requests that affect a varying number of logical blocks. Such large writes would translate into several single block inserts and lookups if a hash table implementation is used.

In addition to the Metadata Remapper, a *remap bitmap* is maintained to keep track of free and used blocks on the available physical device; the remap bitmap supports allocation both of a single remapped block and a range of remapped blocks. The destination

(or remapped) location for a request is determined using a simple algorithm which takes as input the number of contiguous blocks that need to be remapped and finds the first available chunk of space from the *remap bitmap*. By allowing an arbitrary range of blocks to be remapped together, the sequentiality of the blocks is preserved which is beneficial when a disk is used as the backing store. This can be done statically or at runtime; for the Ext3 file system, since most of the blocks are statically allocated, the remapping for these blocks can also be done statically to improve performance. Subsequent writes to other dynamically allocated metadata blocks are remapped dynamically. From our experience, this simple algorithm lays out blocks on disk quite efficiently. More sophisticated allocation algorithms based on the locality of reference can be implemented in the future.

## 2.4 Evaluation

We seek to answer three important questions. First, how accurately does David predict benchmark runtime and what storage space savings does it provide? Second, can David scale to large target devices including RAID? Finally, what is the memory and CPU overhead of David?

### 2.4.1 Experimental Platform

We have developed David for the Linux operating system. The hard disks currently modeled are the 1 TB Hitachi HDS721010KLA330 (referred to as $D_{1TB}$) and the 80 GB Hitachi HDS728080PLA380 (referred to as $D_{80GB}$); table 2.1 lists their relevant parameters. Unless specified otherwise, the following hold for all the experiments: (1) machine used has a quad-core Intel processor and 4GB RAM running Linux 2.6.23.1 (2) Ext3 file system is mounted in ordered-journaling mode with a commit interval of 5 sec (3) microbenchmarks were run directly on the disk without a file system (4) David predicts

| Parameter | H1 | H2 |
|---|---|---|
| Disk size | 80 GB | 1 TB |
| Rotational Speed | 7200 RPM | 7200 RPM |
| Number of cylinders | 88283 | 147583 |
| Number of zones | 30 | 30 |
| Sectors per track | 567 to 1170 | 840 to 1680 |
| Cylinders per zone | 1444 to 1521 | 1279 to 8320 |
| On-disk cache size | 2870 KB | 300 MB |
| Disk cache segment | 260 KB | 600 KB |
| Req scheduling† | FIFO | FIFO |
| Cache segments | 11 | 500 |
| Cache R/W partition | Varies | Varies |
| Bus Transfer | 133 MBps | 133 MBps |
| Seek profile(long) | 3800+(cyl*116)/$10^3$ | 3300+(cyl*5)/$10^6$ |
| Seek profile(short) | 300+$\sqrt{(cyl * 2235)}$ | 700+$\sqrt{cyl}$ |
| Head switch | 1.4 ms | 1.4 ms |
| Cylinder switch | 1.6 ms | 1.6 ms |
| Dev driver req queue† | 128-160 | 128-160 |
| Req queue timeout† | 3 ms (unplug) | 3 ms (unplug) |

Table 2.1: **Storage Model Parameters in David.** *Lists important parameters obtained to model disks Hitachi HDS728080PLA380 (H1) and Hitachi HDS721010KLA330 (H2). †denotes parameters of I/O request queue (IORQ).*

the benchmark runtime for a target $D_{1TB}$ while in fact running on the available $D_{80GB}$ (5) to validate accuracy, the benchmark application was instead run directly on $D_{1TB}$.

### 2.4.2 David Accuracy

Next, we want to measure how accurately David predicts the benchmark runtime. Table 2.3 lists the accuracy of David for a variety of benchmark applications for both Ext3 and Btrfs. We have chosen a set of benchmarks that are commonly used and also stress various paths that disk requests take within David. The first and second columns of Table 2.2 show the storage space consumed by the benchmark workload without and with David. The third column shows the percentage savings in storage space achieved by using David. The first and second columns in Table 2.3 shows the

| Benchmark Workload | Original Storage | David Storage | Storage Savings (%) |
|---|---|---|---|
| mkfs | 931.5 GB | 7.5 GB | 99.19 |
| imp | 10.7 GB | 17.9 MB | 99.84 |
| tar | 20.6 MB | 628 KB | 97.03 |
| postmark | 199.8 MB | 404 KB | 99.80 |
| webserver | 3.7 GB | 3.8 MB | 99.89 |
| varmail | 7.7 MB | 3.8 MB | 50.07 |

Table 2.2: **David's Storage Savings.** *Shows savings in capacity achieved by David. Webserver and varmail workloads are generated using the FileBench benchmarking tool [124].*

| Benchmark Workload | Implicit I/O Classification (Ext3) | | | Explicit I/O Classification (Btrfs) | | |
|---|---|---|---|---|---|---|
| | Original Runtime (Secs) | David Runtime (Secs) | Runtime Error (%) | Original Runtime (Secs) | David Runtime (Secs) | Runtime Error (%) |
| mkfs | 278.66 | 281.81 | 1.13 | 0.228 | 0.049 | - |
| imp | 344.18 | 339.42 | -1.38 | 327.294 | 324.057 | 0.99 |
| tar | 257.66 | 255.33 | -0.9 | 146.472 | 135.014 | 7.8 |
| grep | 250.52 | 254.40 | 1.55 | 141.960 | 138.455 | 2.47 |
| virus scan | 55.60 | 47.95 | -13.75 | 27.420 | 31.555 | 15.08 |
| find | 26.21 | 26.60 | 1.5 | 0.341 | 0.514 | - |
| du | 102.69 | 101.36 | -1.29 | 0.222 | 0.474 | - |
| postmark | 33.23 | 29.34 | -11.69 | 22.709 | 22.243 | 2.05 |
| webserver | 127.04 | 126.94 | -0.08 | 125.611 | 126.504 | 0.71 |
| varmail | 126.66 | 126.27 | -0.31 | 126.019 | 126.478 | 0.36 |

| Run directly on the disk (without a file system) | | | |
|---|---|---|---|
| Benchmark Workload | Original Runtime (Secs) | David Runtime (Secs) | Runtime Error (%) |
| sr | 40.32 | 44.90 | 11.34 |
| rr | 913.10 | 935.46 | 2.45 |
| sw | 57.28 | 58.96 | 2.93 |
| rw | 308.74 | 291.40 | -5.62 |

Table 2.3: **David's Accuracy.** *Shows accuracy of runtime prediction, and the overhead of storage modeling for different workloads. Webserver and varmail workloads are generated using the FileBench benchmarking tool [124]; virus scan using AVG antivirus software.*

original benchmark runtime without David on $D_{1TB}$ and the benchmark runtime with David on $D_{80GB}$ respectively. The third sixth column shows the percentage error in the prediction of the benchmark runtime by David. The first three columns are for the implicit I/O classification approach using the Ext3 file system. The final three columns in Table 2.3 show the original and modeled runtime, and the percentage error when using the explicit I/O classification approach with the Btrfs file system. The storage space savings are roughly the same for Ext3 and Btrfs file systems. The *sr*, *rr*, *sw*, and *rw* workloads are run directly on the raw device and hence are independent of the file system.

*mkfs* creates a file system with a 4 KB block size over the 1 TB target device exported by David. This workload only writes metadata and David remaps writes issued by *mkfs* sequentially starting from the beginning of $D_{80GB}$; no data squashing occurs in this experiment.

*imp* creates a realistic file-system image of size 10 GB using the publicly available Impressions tool [4]. A total of 5000 regular files and 1000 directories are created with an average of 10.2 files per directory. This workload is a data-write intensive workload and most of the issued writes end up being squashed by David.

*tar* uses the GNU tar utility to create a gzipped archive of the file-system image of size 10 GB created by *imp*; it writes the newly created archive in the same file system. This workload is a data read and data write intensive workload. The data reads are satisfied by the Data Generator without accessing the available disk, while the data writes end up being squashed.

*grep* uses the GNU grep utility to search for the expression "nothing" in the content generated by both *imp* and *tar*. This workload issues significant amounts of data reads and small amounts of metadata reads. *virus scan* runs the AVG virus scanner on the file-system image created by *imp*. *find* and *du* run the GNU find and GNU du utilities over the content generated by both *imp* and *tar*. These two workloads are metadata read

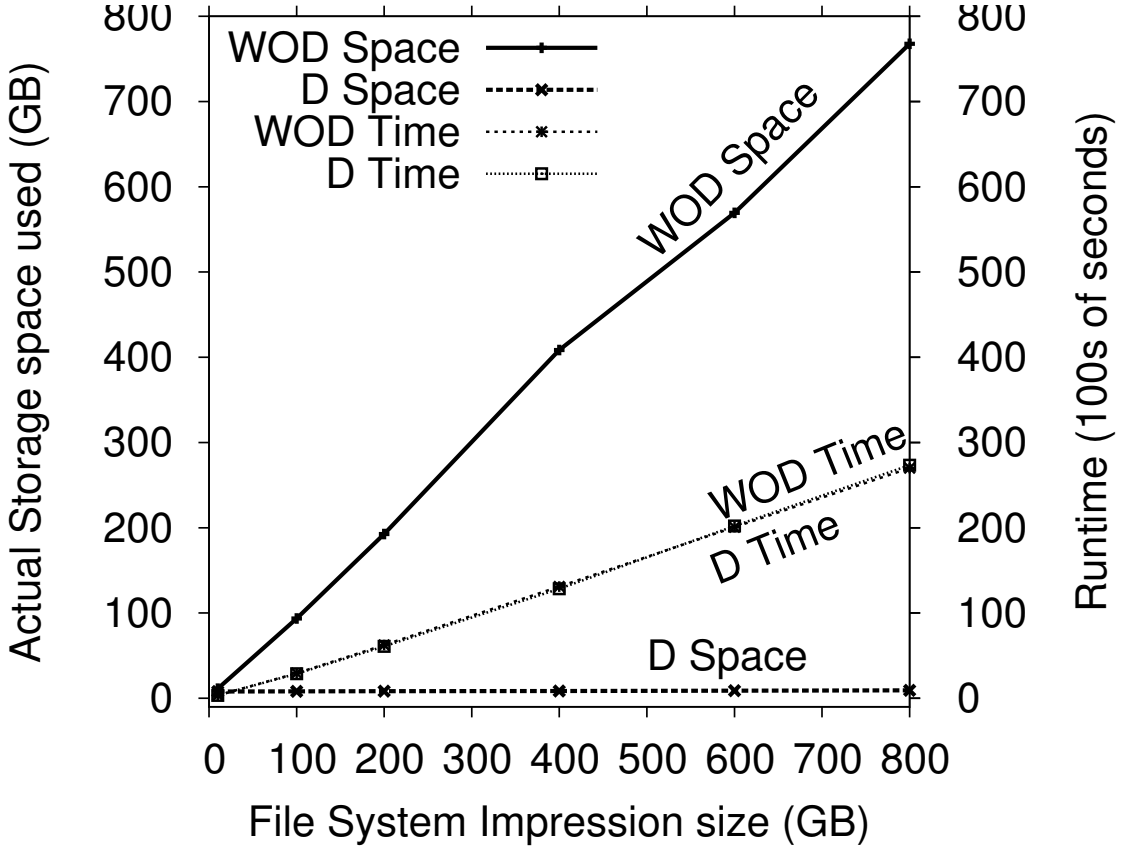



Figure 2.3: **Storage Space Savings and Model Accuracy.** *The "Space" lines show the savings in storage space achieved when using David for the* impressions *workload with file-system images of varying sizes until 800GB; "Time" lines show the accuracy of runtime prediction for the same workload.* **WOD***: space/time without David,* **D***: space/time with David.*

only workloads.

David works well with both the implicit and explicit I/O techniques demonstrating its usefulness across two very different file systems. Table 2.2 shows how David provides tremendous savings in the required storage capacity, upwards of 99% (a 100-fold or more reduction) for most workloads. David also predicts benchmark runtime quite accurately. Prediction error for most workloads is less than 3%, although for a few it is just over 10%. The errors in the predicted runtimes stem from the relative simplicity of our in-kernel Disk Model; for example, it does not capture the layout of physical blocks on the magnetic media accurately. This information is not published by the disk manufacturers and experimental inference is not possible for ATA disks that do not

have a command similar to the SCSI `mode page`.

### 2.4.3 David Scalability

David is aimed at providing scalable emulation using commodity hardware; it is important that accuracy is not compromised at larger scale. Figure 2.3 shows the accuracy and storage space savings provided by David while creating file-system images of 100s of GB. Using an available capacity of only 10 GB, David can model the runtime of Impressions in creating a realistic file-system image of 800 GB; in contrast to the linear scaling of the target capacity demanded, David barely requires any extra available capacity. David also predicts the benchmark runtime within a maximum of 2.5% error even with the huge disparity between target and available disks at the 800 GB mark, as shown in Figure 2.3.

The reason we limit these experiments to a target capacity of less than 1 TB is because we had access to only a terabyte sized disk against which we could validate the accuracy of David. Extrapolating from this experience, we believe David will enable one to emulate disks of 10s or 100s of TB given the 1 TB disk.

### 2.4.4 David for RAID

David can also provide effective RAID emulation. To demonstrate simple RAID configurations with David, each component disk is emulated using a memory-backed "compressed" device underneath software RAID. David exports multiple block devices with separate major and minor numbers; it differentiates requests to different devices using the major number. For the purpose of performance benchmarking, David uses a single memory-based backing store for all the compressed RAID devices. Using multiple threads, the Storage Model maintains separate state for each of the devices being emulated. Requests are placed in a single request queue tagged with a device identifier; individual Storage Model threads for each device fetch one request at a time

| Num Disks | Random Reads | Random Writes | Sequential Reads | Sequential Writes |
|---|---|---|---|---|
| Measured | | | | |
| 3 | 232.77 | 72.37 | 119.29 | 119.98 |
| 2 | 156.76 | 72.02 | 119.11 | 119.33 |
| 1 | 78.66 | 71.88 | 118.65 | 118.71 |
| Modeled | | | | |
| 3 | 238.79 | 73.77 | 119.44 | 119.40 |
| 2 | 159.36 | 72.21 | 119.16 | 119.21 |
| 1 | 79.56 | 72.15 | 118.95 | 118.83 |

Table 2.4: **David Software RAID-1 Emulation.** *Shows IOPS for a software RAID-1 setup using David with memory as backing store; workload issues 20000 read and write requests through concurrent processes which equal the number of disks in the experiment. 1 disk experiments run w/o RAID-1.*

from this request queue based on the device identifier. Similar to the single device case, the servicing thread calculates the time at which a request to the device should finish and notifies completion using a callback after a sufficient delay.

David currently only provides mechanisms for simple software RAID emulation that do not need a model of a software RAID itself. New techniques might be needed to emulate more complex commercial RAID configurations, for example, commercial RAID settings using a hardware RAID card.

We present a brief evaluation and validation of software RAID-1 configurations using David. Table 2.4 shows a simple experiment where David emulates a multi-disk software RAID-1 (mirrored) configuration; each device is emulated using a memory-disk as backing store. However, since the multiple disks contain copies of the same block, a single physical copy is stored, further reducing the memory footprint. In each disk setup, a set of threads which equal in number to the number of disks issue a total of 20000 requests. David is able to accurately emulate the software RAID-1 setup upto 3 disks; more complex RAID schemes are left as part of future work.

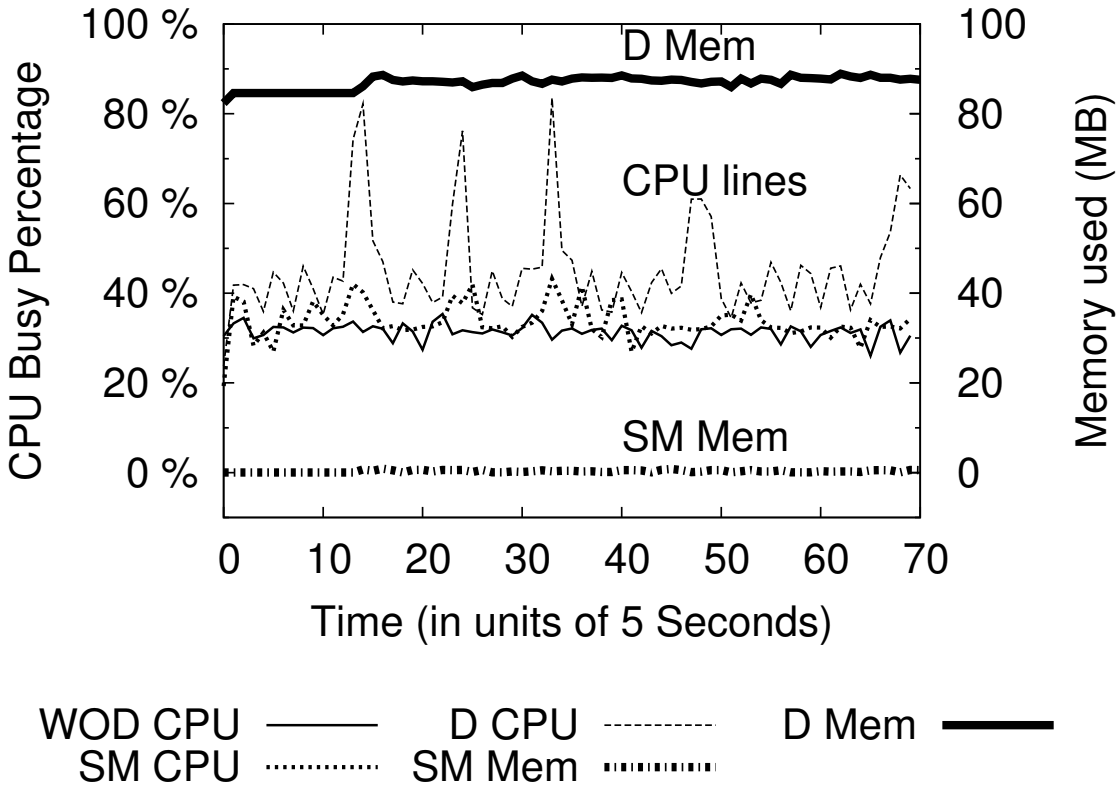



Figure 2.4: **David CPU and Memory Overhead.** *Shows the memory and percentage CPU consumption by David while creating a 10 GB file-system image using* impressions. **WOD CPU***: CPU without David,* **SM CPU***: CPU with Storage Model alone,* **D CPU***: total CPU with David,* **SM Mem***: Storage Model memory alone,* **D Mem***: total memory with David.*

### 2.4.5 David Overhead

David is designed to be used for benchmarking and not as a production system, thus scalability and accuracy are the more relevant metrics of evaluation; we do however want to measure the memory and CPU overhead of using David on the available system to ensure it is practical to use. All memory usage within David is tracked using several counters; David provides support to measure the memory usage of its different components using `ioctls`. To measure the CPU overhead of the Storage Model alone, David is run in the *model-only* mode where block classification, remapping and data squashing are turned off.

In our experience with running different workloads, we found that the memory and CPU usage of David is acceptable for the purposes of benchmarking. As an example, Figure 2.4 shows the CPU and memory consumption by David captured at 5 second intervals while creating a 10 GB file-system image using Impressions. For this experiment, the Storage Model consumes less than 1 MB of memory; the average memory consumed in total by David is less than 90 MB, of which the pre-allocated cache used by the Journal Snooping to temporarily store the journal writes itself contributes 80 MB. Amount of CPU used by the Storage Model alone is insignificant, however implicit classification by the Block Classifier is the primary consumer of CPU using 10% on average with occasional spikes. The CPU overhead is not an issue at all if one uses explicit I/O classification.

## 2.5 Summary

We began this chapter by motivating the need for a better tool to benchmark existing and futuristic storage devices with huge capacity. The storage technology trend over the past few decades clearly shows the continuous increase in storage capacities [91, 121] as well as the introduction of a variety of new physical storage mediums including Phase Change Memory [221, 231], Non Volatile Memory [223] and Shingled Magnetic Disks [1, 2].

David allows a storage researcher or developer to make informed design choices guided by benchmarking results. The key observation behind David is that the benchmark applications do not use file content that is the same as their corresponding real applications but instead only stress the disk with the same I/O pattern as the real application.

We then discussed the details of David's design including it's five core components: block classifier, metadata remapper, data squasher, data generator and the storage

model. The block classifier enables David to classify file system metadata from the benchmark application's data. The metadata remapper component remaps and lays out the metadata writes on the available disk in a space efficient manner. David stores the file system metadata writes on disk but throws away the data writes. The data squasher component is responsible for filtering out the benchmark application's data writes from reaching the disk. The data generator generates fake content when data blocks are read by the benchmark application. It is crucial for David to report accurate benchmark runtime despite throwing away data blocks and remapping metadata blocks. This is where the storage model component helps by accurately modeling the latency of each I/O request in the emulated disk. David responds back to the I/O request after the time reported by the storage model.

We implemented two different non-invasive I/O classification techniques in the block classifier component. The first technique, called "Explicit I/O classification", is borrowed from past research work [3, 189]. We discuss the memory challenge that arises with this technique when scaling it to emulate huge disks and provide a solution. The second technique, called "Implicit I/O classification", is a novel contribution in this dissertation. It can work with many different file systems without significant additional effort.

We then detail the thorough evaluation of David. First, we empirically measure its emulation accuracy for a variety of micro and macro benchmarks. We then test David's ability to accurately emulate large storage disks using minimal physically available disk space. We also show that David can emulate multiple storage disks simultaneously by using the emulated disks in a software RAID-1 array.

# Chapter 3

# Improving Virtualized Storage Performance

Virtual machine monitors (VMMs) have become ubiquitous in the past decade. They form an integral part of the cloud computing infrastructure. In this chapter, we develop a novel non-invasive I/O classification technique using the hardware virtualization extensions in modern processors. As we show later in this chapter, this technique is easy to adopt, deploy and maintain when compared to other approaches that need interface changes. We then build smart caching and deduplication systems in the hypervisor using this I/O classification technique.

VMMs offer several important advantages over more traditional approaches, including server consolidation [115, 163], reduction in infrastructure costs [216], simpler failure handling [116], ease of management [142], support for legacy applications [34, 162, 219], improved security [53, 64, 65, 97, 152], and better reliability [32, 49]. Virtualized storage, found within said VMMs, adds the benefits of storage consolidation, shared storage across VMs, out-of-box support across several guest OSes, reduction of costs, improved availability, efficient backups and quick snapshots [132, 151, 180, 208, 215, 217]. Not surprisingly, both server and storage virtualization are prominent and together form a

central part of all modern cloud computing infrastructures.

As the lowest level in the software stack, the VMM [15] must manage system resources, including memory, disk, CPU, and network. In doing so, the VMM must optimize their usage for high performance, fairness, and other important system-wide goals.

Managing resources effectively fundamentally requires *information:* which I/O request is latency sensitive, and thus should be scheduled soon? Which block is likely to be accessed again soon, and thus would benefit from placement within a cache? Without this type of information, making the decisions a resource manager must make are at best arduous and often impossible. For example, a VMM cannot typically differentiate whether an I/O request consists of application data or is file-system journaling traffic. Without such basic knowledge, the VMM is inherently limited in its resource-management capacity.

The main hypothesis that underlies this chapter is that the VMM can efficiently gain access to a wealth of important and necessary information through judicious usage of *system-call interposition*. In such a configuration, operating system level system call entry and exit are routed through the VMM. At these critical junctures, the VMM can record relevant pieces of information as well as take necessary actions in order to gain access to facts pertinent to its operation.

To explore this hypothesis, we have designed and implemented Sky, a prototype VMM with system-call interception at its core.[1] Sky is implemented for the x86-64 architecture and it supports Linux and FreeBSD operating systems. Sky extends KVM with system-call interception to facilitate a range of new information-gathering techniques. Specifically, Sky implements a core interception framework to track specific processes and threads, and then obtains storage-specific insights atop this basic machinery. The insights include information such as the size of currently accessed files, the classification of

[1]The acronym for System Call Interception, SCI, and one possible pronunciation, motivates our name.

block I/O into data and metadata, and file content assessment. Some of this information is approximate (i.e., not guaranteed to be correct); however, as we show, it is still useful in building various storage-system optimizations. To aid its information gathering, Sky also (on occasion) injects its own system calls into the OS above; said *insightcalls* are a useful general knowledge-acquisition technique atop the base interception mechanism.

We demonstrate the utility of Sky by implementing three case studies, each showcasing different possibilities within the Sky infrastructure. The first is a simple monitoring tool (Section 3.5), which can provide generic information such as block lifetimes and the amount of metadata generated by different file systems. With such monitoring in place, a VMM can serve as a single point of detailed knowledge about guest file-system behavior.

The second case study, which we refer to as iCache (Section 3.6), implements an aggressive VMM-level caching policy [131], leading to a 2.3 to 8.8 times improvement in run time for both search and database workloads. This approach gives higher priority to small files and file-system metadata and thus can improve run-time significantly. We also show how an application (the MySQL database server) can provide further hints to the caching layer via Sky and improve performance further.

The third case study, known as iDedup (Section 3.7), takes advantage of Sky's file-content information to improve performance of a block-layer deduplication system [206]. Sky provides hints to iDedup about block usage patterns (Section 3.2.5), and iDedup uses such hints to avoid expensive lookups and thus improves performance. Specifically, this optimization reduces run time by 4.5 to 18.7 times for file-copy and encryption workloads (Section 3.7.2).

In each of these cases, Sky implements improvements within a VMM that previously had required full-stack modifications to obtain the information needed to implement said functionality. Sky, in contrast, functions across operating systems (Linux and FreeBSD), and different file systems (Linux Ext4, Btrfs, and XFS, for example). In this

| Compared Research Work | Number of modified components (Names of the supported components) | | |
|---|---|---|---|
| | **OS** | **FS** | **Storage Interfaces** |
| Differentiated Storage Services [131] | 2 (Linux, Windows) | 2 (Ext3,NTFS) | 3 (VFS,Block I/O, iSCSI) |
| Deduplication with hints [122] | 1 (Linux) | 4 (Ext2/3/4,Nilfs2) | 2 (VFS,Block I/O) |
| Sky | **0** (Linux, FreeBSD) | **0** (All)†‡ | **0** (All)† |

Table 3.1: **Ease of Adoption.** *This table compares the number of components of various types that are supported by Sky and other relevant past research work without any additional implementation effort. † Sky currently supports file systems that do not change user-supplied content (e.g., due to compression or encryption within the file system). Sky uses system-call interception and therefore is not affected by the choice of the storage layers (e.g., file system and device drivers) or their interfaces (e.g., VFS, Block I/O and SCSI) present in-between the system call and the VMM. ‡ We have tested Sky with the file systems UFS, ZFS in FreeBSD guest operating system and with Ext3, Ext4, XFS, JFS, Nilfs2, Reiserfs, and Btrfs in Linux guest operating system.*

manner, Sky consolidates implementation of its optimizations, instead of replicating such effort across different file systems and operating systems. We believe that this will make Sky easier to adopt, deploy and maintain in existing systems.

The rest of this chapter is structured as follows. We first provide further motivation (Section 3.1). We then describe the design (Section 3.2) and implementation (Section 3.3) of Sky. Finally, we evaluate Sky (Sections 3.4 to 3.8), discuss related work (Section 5.2), and summarize in (Section 3.10).

## 3.1 Motivation

In modern virtualized storage systems, better performance, quality of service, and other critical optimizations and features can be achieved through access to information. For example, previous works have shown that classifying I/O requests, and treating each

class differently, can greatly improve performance for some workloads [122, 131, 208].

Unfortunately, due to the simple, restrictive interfaces exposed by each of the layers, information cannot be easily passed through the many layers of the storage stack. This reality leads to the so-called "semantic gap" [41] across said layers, thus leading to many missed opportunities in the storage stack [54, 81, 129, 131, 160, 171, 185, 186, 188, 189, 225].

Many examples exist in the literature that showcase the benefits of information (and control) throughout the storage stack. For example, Thereska et al. classify I/O requests into flows and associate policies for each of the I/O flows to allow differentiated treatment [208]. Mesnier et al. explicitly classify I/O requests to improve performance, by modifying the application, file system, and low-level storage interfaces [131]. Sonam et al. use I/O classification to improve inline block-layer deduplication by modifying the application, file system, and block I/O interface to generate hints about file-system metadata and file contents [122]. All approaches require changes across many layers of the storage stack.

While these systems all provide significant benefits, we believe there are important reasons that they often do not reach deployment. One prominent reason is that any idea that must be realized throughout the storage stack creates a large burden upon developers. Table 3.1 compares the number of different types of operating systems, file systems, and storage interfaces that must be modified to support various storage optimizations [122, 131]. Being able to run underneath multiple operating systems, file systems, and interfaces has a multiplicative effect on developers, who must modify each of these components to reach wide-scale deployment. In contrast, as the table shows, Sky works across different file systems and storage interfaces, and provides the infrastructure needed to work underneath Linux and FreeBSD; developers of new storage optimizations can thus implement them once within the Sky framework and deploy them underneath a wide range of systems.

Of course, if all vendors agree on a set of information to pass across layers, it is possible that new standards could be developed and adopted. However, as others have discussed, changing interfaces is difficult and time consuming [186]; even small changes to the low-level disk standards, such as the evolution from block-based to object-based storage [67], may take many years to come to fruition, or never reach wide-scale adoption at all.

The best system to support cross-layer optimizations requires modification only at a single spot in the stack, not requiring changes throughout many layers. The optimizations realized in such a framework should then work across a broad range of systems with little or no effort. We now describe one such system that we have built, Sky, which is implemented as part of the VMM and enables interesting information-based storage services to be realized.

## 3.2 Design

This section describes the basic techniques used by Sky (implemented as part of the VMM) to intercept system calls (Subsections 3.2.1 and 3.2.2) and then details the insights gained by intercepting I/O-related system calls (Subsections 3.2.3 to 3.2.5). Our design was influenced by the following desirables:

- *Simple and Universal:* Favor simple techniques that are widely applicable across operating systems.
- *Timely:* Generate reliable hints as early as possible.
- *Robust and Lightweight:* Keep Sky robust and its overheads low.

Table 3.2 is a summary of all the information tracked by Sky and where the information is used. Sky tracks information about processes, threads, process parent-child relationship and in-progress system calls in order to provide the basic interception framework upon which meaningful insight gathering techniques can be implemented.

| Tracked Information | Used for |
|---|---|
| List of monitored processes, their PIDs and their page directory base address. | Interception Framework (Subsection 3.2.1) |
| Threads of monitored processes, their TIDs, stack base pointers and stack size. | Interception Framework (Subsection 3.2.1) |
| Parent child relationship between monitored processes. | Interception Framework (Subsection 3.2.1) |
| System Calls in progress for monitored processes along with their arguments and userspace stack pointer value. | Interception Framework (Subsection 3.2.1) |
| A per-process pool of 8KB userspace buffers allocated by Sky for issuing insightcalls that need their arguments to be in memory. | Interception Framework (Subsection 3.2.2) |
| List of monitored file descriptors, their current file offsets and the maximum file offset accessed so far. | Insight I (Subsection 3.2.3) |
| List of memory-mapped pages for monitored files in monitored processes and their corresponding guest-physical addresses. | Insight II (Subsection 3.2.4) |
| Checksums of 4096-byte chunks in data payload of I/O-related system calls are stored while the system call is getting processed. Checksums of 4096-byte chunks in data read from or written to the virtual disk are retained for a certain time period. | Insight II (Subsection 3.2.4) |
| Whether a process has file-copy I/O pattern or is encrypting files. Detecting file-copy I/O pattern requires storing checksums of 4096-byte chunks of data read or written by applications temporarily. | Insight III (Subsection 3.2.5) |
| The checksums of 4096-byte chunks at various file offsets when block lifetimes need to be calculated. | Block Lifetime (Subsection 3.5.2) |

Table 3.2: **Information tracked by Sky.** *This table lists the information tracked by Sky about the guest operating system and its processes.*

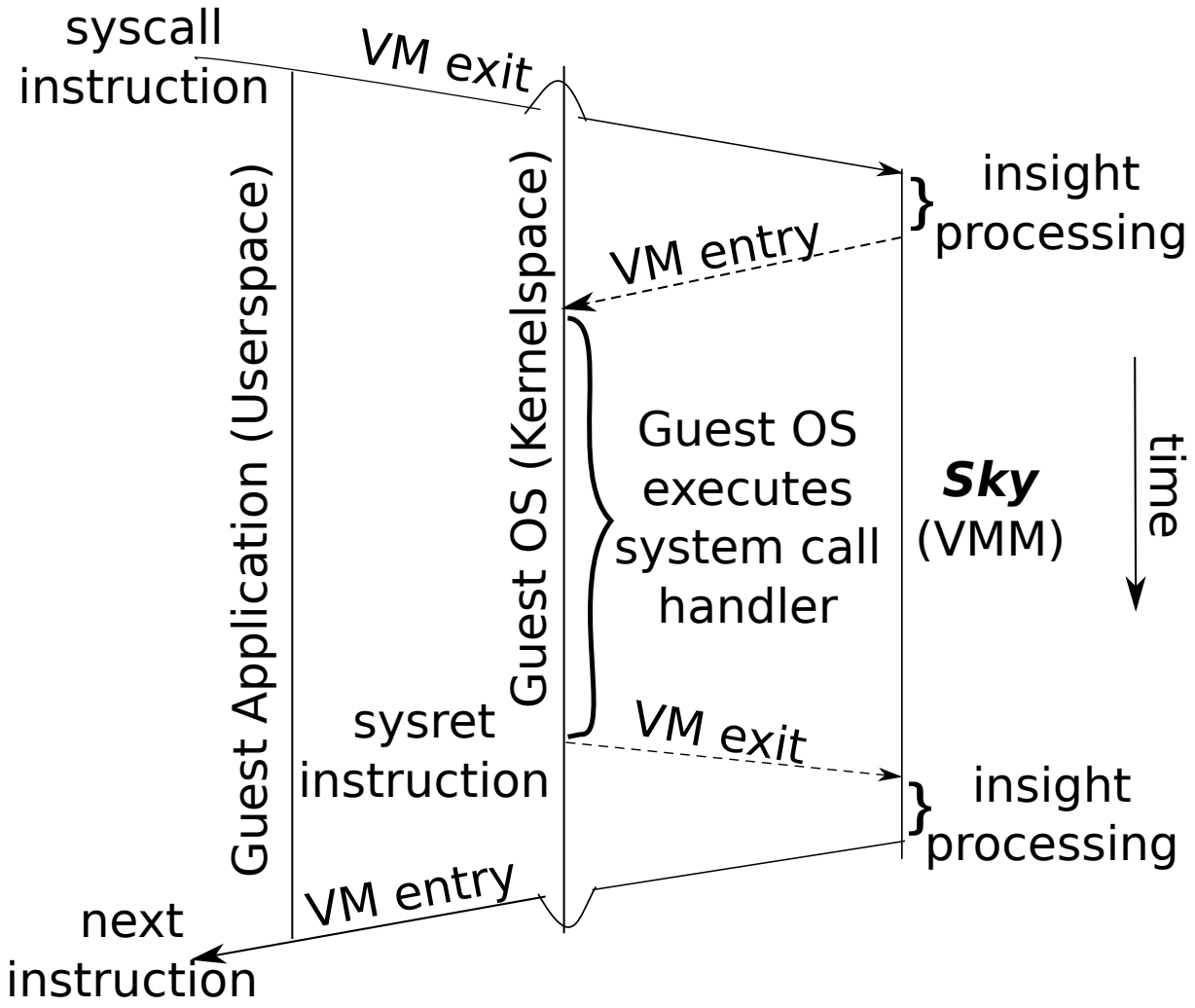



Figure 3.1: **System-Call Interception.** *This figure shows the control flow between the guest application, guest operating system and the VMM during a system-call interception in a monitored process. Sky turns off the hardware interception techniques when unmonitored processes are scheduled on a processor.*

Depending on the insight generated, additional information is tracked by Sky as listed in Table 3.2.

### 3.2.1 System-Call Interception

Sky intercepts the entry and exit of a subset of I/O and process-management related system calls executed by the guest application in order to gain insights that can be used as hints to improve virtualized-storage performance. Sky configures the processor

before a VM entry so that execution of a system call entry or exit instruction causes a VM exit and transfers control back to Sky (part of VMM). Figure 3.1 shows the control flow during system-call interception. With this ability to intercept system call entries and exits, Sky can monitor the arguments and return codes of system calls to gather insights about the guest application. Being part of the VMM, Sky can access all of the guest VM's memory enabling further optimizations and new features.

**Selective System-Call Interception**

Sky avoids the overheads due to intercepting all userspace applications in the guest VM by monitoring only a targeted set of I/O-bound processes. Sky automatically monitors and unmonitors the children of the monitored processes by intercepting process-management related system calls like `fork`, `clone` and `kill`. Sky monitors all threads in a monitored process by default. Whenever a new guest process is scheduled on a virtual processor, Sky checks if the new process is monitored or not and turns system-call interception on or off respectively.

In our prototype, monitoring of applications is bootstrapped by a helper application calling a library function with its own PID and then launching the benchmark application. The library function sends the PID to Sky (which is part of the VMM in the host machine) through the network. Sky automatically monitors the launched benchmark application and any other processes it subsequently creates. More sophisticated policies could be built on top of this scheme when necessary in the future: e.g. tracking and identifying certain applications that are known beforehand to benefit from Sky or periodically monitoring the latency of I/O-related system calls made by guest applications and dynamically turning system-call interception on or off based on these latencies. Our prototype version does not do this.

**Identifying Processes and Threads:** Sky keeps track of the guest operating system assigned process identifiers (PIDs) and thread identifiers (TIDs) for the monitored processes and threads respectively. When intercepting system call entries and exits, Sky uses only the virtual-CPU state to identify the currently executing process or thread. Specifically, processes are identified using the page directory base register (PDBR) that contains the guest-physical address of the currently executing process's page directory base. Sky maps a PID to the page directory base address by issuing a `getpid` insight-call (described in Subsection 3.2.2). Sky captures the userspace stack base address, stack size and the thread identifier while intercepting thread-creation system calls. Sky differentiates between threads within a process using the stack pointer (SP) register that contains the guest-physical address of the top of the userspace stack. Given the values of the PDBR and SP register, Sky identifies the guest operating system issued process and thread identifiers respectively. Sky maintains a set of all monitored processes, their PIDs, PDBRs, threads, thread TIDs, thread SP values and thread stack sizes.

**Tracking Guest Operating System Scheduling:** Sky intercepts all guest operating system process scheduling by intercepting overwrites to the PDBR of the virtual processor through hardware mechanisms. The PDBR has to be compulsorily overwritten with the page directory base address of the new process during process context switch. Since thread rescheduling does not involve a PDBR overwrite, Sky uses the following technique: during a system-call interception, if Sky detects a different currently running TID from the one that last executed on the same virtual processor during the last system-call interception, it knows a thread switch has occurred. Such delayed detection of a thread switch only when a system call is intercepted is sufficient for matching a system call exit correctly with its entry.

### 3.2.2 Insight-Calls: Sky-Introduced System Calls

In certain scenarios, Sky needs access to the state maintained by the guest operating system in order to gather more insights efficiently, easily and in a manner that eases portability across different guest operating systems. Sky (which is part of the VMM) issues system calls to the guest operating system in the context of an intercepted guest application in order to read state from the guest operating system. We call such Sky-issued system calls as Insight-Calls. Sky currently issues insight-calls only when it is intercepting an actual system call made by a guest application and to only read state from the guest operating system. Insight-calls never change the state of the guest operating system because that would be outside the knowledge of the guest application and could lead to erroneous application behavior.

**Insight-Call:** To issue an insight-call, Sky first saves the intercepted system call entry's system call number and arguments (call it *"syscallinformation"*) into a private data structure and then replaces them with those corresponding to the insight-call that it wishes to issue to the guest operating system. When Sky subsequently intercepts the system call exit, it restores back *"syscallinformation"* into the appropriate registers and additionally decrements the current instruction pointer (IP) appropriately to point back to the system call entry instruction. This way, the original guest-issued system call is now executed by the guest operating system. Sky can also issue a series of such insight-calls when more complex information needs to be gathered from the guest operating system. Sky uses insight-calls in several scenarios like: getting the PID of the process currently executing (Subsection 3.2.1), getting the current size of the file backing an open file descriptor (Subsection 3.2.3) and handling "misaligned or small I/O requests" (Subsection 3.2.4).

We note that, customers who do not completely trust their service providers (e.g., in a IaaS cloud computing model) with the usage of insight-calls could be given an option to opt-out of Sky during their sign-up process. Also, to improve performance

in some cases, it is possible to avoid insight-calls and instead directly access the guest operating system internal state to get the required information [55]. Sky does not use such optimizations because the effort does not seem justified given the relatively small number of insight-calls. Exploring such optimizations is left as future work.

### 3.2.3 Insight I: Guest File System File Size Information

A storage system cache can achieve improved cache hit rates by knowing whether an I/O request is issued on a small file or a large file [131]. This is because large files are usually laid out sequentially on a magnetic disk and therefore cache misses on reads to small files are costlier than misses on a large file. Moreover, more small files can be cached in the same cache space occupied by a large file. Sky implicitly classifies I/O requests based on the size of the corresponding file by keeping track of the current file sizes of all files opened by a monitored process. An example of such file size based classification is shown later in Table 3.6 of the *iCache* case study (Section 3.6).

Sky achieves such file-size based classification by intercepting the I/O-related system calls like `open`, `read`, `write`, `lseek` and `close` in order to capture information like: current open file descriptors in a process, the current file offsets for those file descriptors, the files behind those file descriptors and their current file sizes.

**Selectively monitoring only certain files:** Sky can be instructed to selectively intercept I/O to only specific files using a control command sent through the network. The current prototype version of Sky allows specifying such files using their path prefix that denotes their location in the guest file-system hierarchy. Sky issues a `getcwd` insight-call in order to get the current working directory of the process before prefix matching files opened using relative file paths. A variety of other policies for specifying which files to selectively monitor are possible. Our current prototype implementation does not handle symbolic or hard links and requires that separate control commands be used to instruct

Sky to monitor such links as well. However, a future version of Sky could use additional insight-calls during file open time to check for and resolve symbolic or hard links to verify if they need to be monitored.

**Tracking File Sizes:** Sky tracks the current file offset and the highest file offset accessed so far for all the monitored file descriptors whenever an I/O is performed by a guest application using read, write and other similar system calls. Sky also tracks the current file size for all monitored file descriptors using an `lstat` or `lseek` insight-call.

Sky translates the file size information into an *I/O class* and hints the VMM-level storage cache to adjust the priority based on the *I/O class*. Sky uses checksums of the payload to match insights obtained by intercepting system calls with their corresponding I/O requests that occur later. Therefore, Sky always associates gathered insights with a particular I/O request rather than with the virtual-disk sectors to which the I/O request is destined. This prevents insights from becoming stale when the corresponding sectors are reallocated to a different file. A case study on such a smart storage cache called *iCache* with its performance compared against a normal storage cache is detailed in Section 3.6.

### 3.2.4 Insight II: Guest File System Metadata vs. Data Classification

Cache hits can be improved by distinguishing file system metadata from application data and giving higher priority to file system metadata [131]. The file system inside the guest operating system kernel organizes information on the virtual disk by writing metadata information (e.g. block allocation bitmap, file offset to disk block translation) in addition to the data from the guest application. However, the distinction between guest file system written metadata and guest-application written data is not available at the virtual disk in the VMM. Sky provides useful hints to the virtual disk to distinguish metadata I/O requests from data I/O requests. The basic idea behind this insight is the observation that all data I/O requests originate from the guest application while

all metadata I/O requests originate from the in-kernel file system within the guest operating system. Sky tracks the set of all data I/O requests that originate from the guest application using system-call interception and identifies metadata I/O requests by exclusion from this set. Subsections 3.2.4 to 3.2.4 detail how to handle different types of I/O system calls using this basic technique.

**Handling Synchronous I/O**

Synchronous I/O is performed primarily using the `read` and `write` system calls. Both take three arguments: the open file descriptor on which I/O is requested, the address of a userspace buffer for I/O contents and the number of bytes in the I/O. These system calls return the number of bytes successfully accessed upon success and a negative error code upon failure. Other system calls for performing synchronous I/O like `pread`, `pwrite`, `preadv` and `pwritev` are handled similarly.

**Writes:** When a guest application issues a `write` system call to write data to a file, Sky intercepts the system call entry and calculates the checksums of every 4096-byte sized chunk in the userspace buffer supplied by the application. Sky, being part of the VMM, easily translates the guest-virtual address of the userspace buffer to host-virtual addresses while accessing the userspace buffer. Sky stores these checksums in a hash table. The *I/O class* based on the file size insight described earlier in Subsection 3.2.3 can also be stored in this hash table. When this system call is then serviced by the guest operating system, it eventually causes a write I/O request to the virtual disk. Sky also interposes on this subsequent write request to the virtual disk to calculate the checksums of every 4096-byte chunk and looks up the checksums in the hash table. If the checksum is found, it indicates that the content originated from the guest application and hence is a data I/O request. Checksums for metadata I/O requests will never be found in the hash table. Sky removes the checksums from the hash table after the lookup to avoid

any future misclassification.

**Reads:** Reads have to be handled slightly differently by Sky. When a guest application issues a `read` system call, the data is available in the userspace buffer only after the system call completes because the data has to be read from the virtual disk or the buffer cache as part of the `read` system call servicing by the guest operating system. Hence, Sky interposes all read requests to the virtual disk and calculates the checksums of every 4096-byte sized chunk being read and stores them in a hash table along with the corresponding sector number in the virtual disk and a timestamp. Subsequently, Sky also interposes the exit of the `read` system call that caused the read request to the virtual disk, calculates the checksums of every 4096-byte chunk in the userspace buffer and looks up the checksums from the hash table. If the lookup succeeds, Sky classifies the request as data I/O and removes the checksums from the hash table. All entries remaining in the hash table after a configurable sufficiently long delay (currently set at 4 seconds) are classified as reads due to metadata I/O requests. In the experiments presented in this chapter, we never had to re-configure this delay value.

**Handling Asynchronous I/O:**

When a guest application issues an asynchronous I/O system call, the I/O is not complete when the system call returns. Rather, the I/O completes at a later point in time and the guest application learns about the completion later using a separate system call. Hence, for asynchronous read I/O system calls, Sky performs the checksum calculation and lookups after the I/O request is completed by the guest operating system. For asynchronous write I/O requests, the checksum calculation occurs during the I/O-submission system call entry.

**Handling Memory Mapped I/O:**

Memory mapped I/O is performed by mapping a region of the file address space to a region of the process's virtual-memory address space. I/O requests to the virtual disk are automatically issued by the guest operating system when the memory mapped address space is accessed by the guest application. Because there are no system calls to intercept for insights when I/O happens, memory-mapped I/O is specially handled by Sky. Sky intercepts the `mmap` system call that initially performs the memory mapping with parameters that specify the starting virtual-memory address and the length of the memory-mapped region. At this time, Sky write protects the host operating system memory pages that contain the guest page-table entries behind the memory-mapped virtual address space. This write protection ensures traps to Sky whenever the guest operating system changes the guest-physical pages backing the memory-mapped virtual address space. Thus, Sky continuously keeps track of the most recent guest-physical pages (and their host-physical pages) backing the memory-mapped virtual address space in a global hash table.

A memory-mapped I/O request automatically issued by the guest operating system to the virtual disk always contains the guest-physical address backing the memory-mapped virtual page that was accessed. This is because memory-mapped I/O skips the buffer cache in the guest operating system. When Sky intercepts the I/O requests to the virtual disk, it also looks up the host-physical addresses of the pages behind every I/O request in the global hash table described above. A successful lookup indicates a data I/O request while a failure means metadata. Sky removes the old addresses and adds new ones to the hash table when the guest operating system unmaps the old guest-physical pages and maps new ones for the memory-mapped virtual address space.

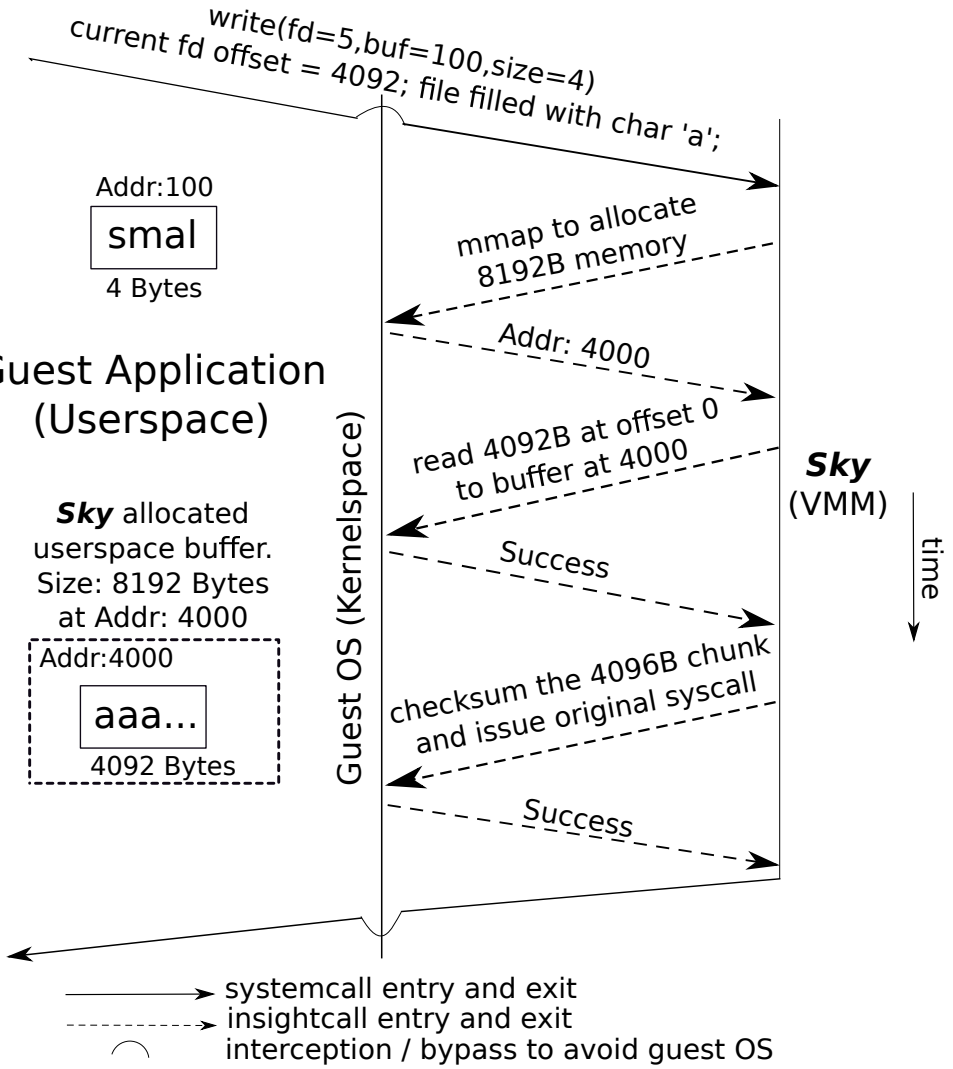



Figure 3.2: **Insight-Calls for handling a Small Write.** *This figure shows how Sky handles an example small write using a series of insight-calls.*

**Handling Misaligned and Small I/O:**

Sky always calculates checksums of 4096-byte chunks that are aligned with 4096-byte file offset boundaries so that the checksums remain valid when interposing the I/O requests to the virtual disk despite the guest operating system splitting or merging I/O requests. 4096 bytes or a sector is the smallest addressable unit for modern disk drives and is smaller than or equal to the file system block size. However, guest applications can issue I/O system calls that result in chunks smaller than 4096 bytes either due to small I/O

requests or due to I/O requests misaligned with the 4096-byte file offset boundaries. Sky handles such I/O requests by reading the necessary contents from the file using Insight-Calls to form 4096-byte chunks as outlined in Figure 3.2 and described below. It should be noted that because our prototype uses the insights as hints for performance improvements, it could skip handling misaligned and small I/O requests.

**1.** Sky allocates a 8192-Byte private anonymous userspace buffer in the guest-application's virtual address space using a `mmap` insight-call. Sky also adds this 8192- byte userspace buffer into a free memory pool that it maintains for every guest process so that future Insight-Calls for this process can reuse the same buffer.

**2.** Sky then reads any necessary additional content located before and after the small or misaligned I/O request's file offset as needed using `pread` insight-call into the first and second 4096 bytes of the userspace buffer allocated in the previous step.

**3.** Sky then calculates checksums of resulting aligned 4096-byte chunks before finally reissuing the original guest-application's system call. For misaligned reads alone, Sky issues the original guest-application's system call before issuing the `pread` insight-calls so as to avoid multiple disk requests for large multiblock reads. This way the total number of disk I/O requests generated remains the same while handling misaligned or small read requests.

As an optimization, Sky also caches the contents of the most recent I/O along with the file descriptor, file offset and data size information for monitored processes. Using this information, misaligned I/O resulting from a strided access pattern that starts at an unaligned offset can be handled without using the process described above that uses additional insight-calls.

**Note:** Certain file systems store both metadata and data in the same virtual-disk block: e.g. Ext4's inline data feature stores tiny files inside the inode structure. Sky classifies

such blocks as metadata. There is a very small window of chance when a monitored guest application could generate data blocks that match the metadata blocks generated by the guest file system within the short duration of the virtual-disk access times for those data blocks. Since, our prototype uses insights as hints, such small chances of misclassification are tolerable.

### 3.2.5 Insight III: Application I/O semantics and patterns

Tracking the I/O semantics and patterns of applications can be helpful in improving their performance. Sky can detect I/O patterns without any modifications to the application or the guest operating system. Sky sends the I/O-pattern insights as hints to the storage system. Example I/O-pattern insights are "knowing when an application is encrypting data" or "knowing when an application is copying data from one file to another". We show how these I/O-pattern insights can be used to improve the performance of a deduplication system in Section 3.7.

**Detecting File Encryption:** Sky uses the names of the executables and the file name extensions of the destination files to derive hints about encryption. Sky issues a `sysctl` insight-call or a `readlink` insight-call to get the name of the executable depending on whether the guest operating system is Linux or FreeBSD respectively. The file name extensions are available as an argument while intercepting `open` system calls. Sophisticated executable identification and file type detection by examining the contents of the executable and destination file is left as future work. Alternate generic approaches could also be explored in future: e.g. Sky could flag processes whose writes repeatedly fail to get deduplicated so as to skip deduplication for future writes from such flagged processes.

**Detecting File Copy:** Sky detects file-copy I/O patterns by first targeting certain guest applications by using the executable names as a hint: e.g. Unix tools like *cp* and *dd*. Sky then stores the checksums of 4096-byte chunks being read by such targeted applications in a hash table. Finally, Sky looks up the checksums of 4096-byte data chunks being written by these applications in the hash table to confirm the file-copy I/O pattern. Repeated matches indicate a file-copy I/O pattern in progress.

### 3.2.6 Application Supplied Insights

Certain applications already perform or can be easily modified to perform better I/O classification because they have the most information about the I/O requests that they issue. The exact policy used for I/O classification depends on the guest application. For example: a cloud file server can associate its premium customers with a higher priority I/O classification ensuring a better quality-of-service, a database server can associate I/O requests to certain data structures like the "secondary index" with lower priority to ensure better overall throughput (Subsection 3.6.2).

Guest applications can pass the I/O classification information on a per-system-call basis by calling an alternate library function that is similar to its counterpart in standard libraries like libc. This alternate function takes an additional last argument for the I/O class. For example, a C application calls `iwrite(file descriptor, buffer, size, ioclass)` library function to perform writes instead of the usual `write(file descriptor, buffer, size)` libc function. The `iwrite` library function issues the `write` system call with two additional last arguments: an additional magic number argument and the I/O classification number. During system-call interception, if Sky sees that a system call has the magic number as the second-to-last argument, then it indicates that the guest application is supplying explicit I/O classification information in the last argument. Therefore, Sky uses the guest application supplied I/O classification and

turns off its own implicit I/O classification based on "file size", "file system metadata vs. data" and "application I/O semantics" for that I/O request. This approach of passing I/O classification along with every system call allows fine granular control on every I/O request rather than over an entire file. Also, it allows passing down the I/O classification information from the guest application to Sky very efficiently in a timely manner. Guest applications that directly access a virtual disk without a file system can also pass I/O classification using this approach.

## 3.3 Implementation

### 3.3.1 Interception Techniques

**Intercepting System Call entries and exits:** Sky uses previously known techniques [53, 152] to intercept all x86-64 system call instructions except the IRET instruction for which we describe a new technique below. All the experiments in this chapter use 64-bit guest operating systems and they run on an Intel processor; therefore, they all used the Syscall, Sysret and IRET instructions for performing system calls. We tested out some of the other previously known interception techniques [53, 152] listed below but did not use them with Sky. A comparison of Sky with related work in the field of Virtual Machine Introspection is in Section 5.2.

- ***INT 80:*** The Interrupt Descriptor Table (IDT) size is restricted to cause faults during software interrupts.
- ***Syscall and Sysret:*** The SCE flag is unset so that the syscall/sysret instruction causes a VM exit.
- ***Sysenter and Sysexit:*** The SYSENTER_CS_MSR machine status register is set to NULL so that the sysenter/sysexit instruction causes a VM exit.
- ***IRET:*** Both Linux and FreeBSD kernels use the IRET instruction for returning from

a system call during slow return scenarios that include situations where userspace signal handlers are invoked before returning from the system call. We could not easily intercept the IRET instruction directly using architectural support with the Intel virtualization extensions unlike the AMD virtualization technology [9, 90].[2] Sky intercepts the system call exits that use IRET instruction using the following technique. Whenever Sky intercepts a system call entry instruction, Sky subtracts the size of the syscall instruction from the userspace IP register and keeps track of any such subtraction made. This subtraction guarantees an interception during the system call exit because the syscall instruction will be re-executed again artificially upon system call exit. There are two possibilities during the subsequent system call exit: the guest operating system either uses the Sysret/Sysexit instruction or uses the IRET instruction. In the former case, Sky intercepts the Sysret/Sysexit instruction using hardware mechanisms, gathers insight and undoes the subtraction because it is no longer needed. In the latter case, though the IRET instruction cannot be intercepted, the artificial re-execution of the syscall instruction will be intercepted by Sky, at which point Sky completes the insight processing and skips executing the artificial syscall instruction.

**Intercepting Guest Operating System Scheduling:** Both Intel and AMD hardware virtualization extensions allow intercepting writes to the PDBR by setting a specific bit in the VM execution control register.

### 3.3.2 Handling Process Rescheduling

Sky gathers insights by analyzing system call arguments and the returned values. However, there are some tricky scenarios that arise when the guest operating system reschedules monitored processes across different virtual processors. In these scenarios, associ-

[2]The Intel manual [90] mentions that a VM exit occurs upon executing an IRET instruction if the "NMI-window exiting", "NMI Blocking", "Virtual NMIs" and "NMI Exiting" control bits are set. Using this technique, a VMM can queue a virtual NMI to a guest and subsequently inject a virtual NMI when the guest is ready after execution of an IRET instruction. However, we have not verified that this technique can be used for intercepting IRET instructions for the purposes of Sky.

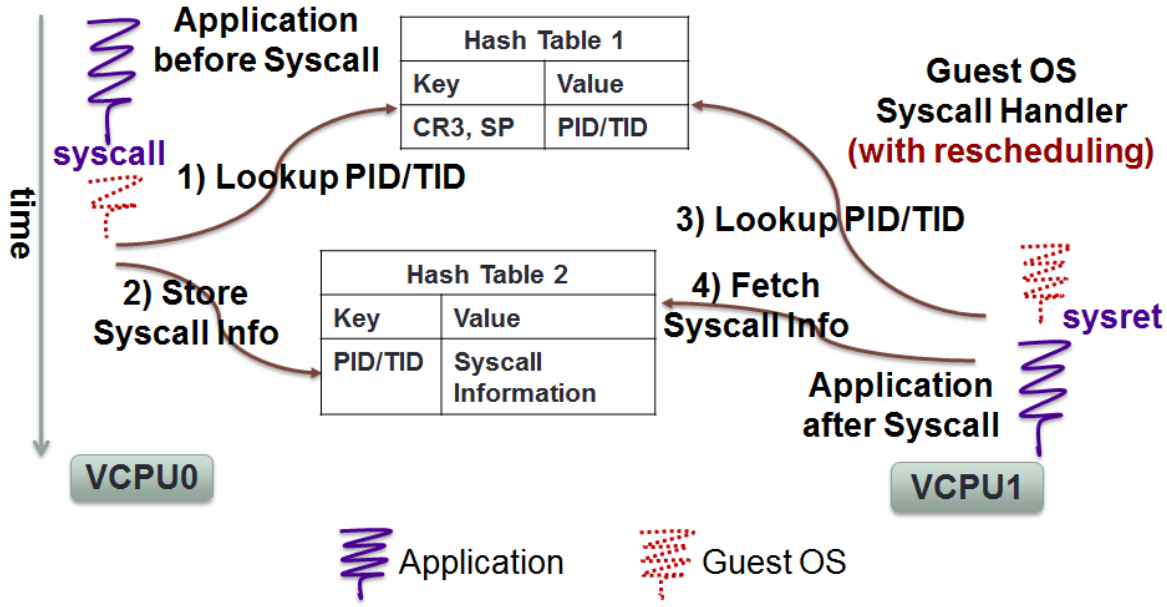



Figure 3.3: **Split System Call.** *This figure depicts how Sky handles the scenario when a system call entry and its exit occur on two different virtual processors due to a context switch during the guest operating system system call handler execution.*

ating the value returned by a system call to its earlier invocation and the corresponding arguments needs additional effort as detailed in the rest of this subsection.

**Matching system call exits with entries:** Sky matches system call exits with entries based on the fact that system calls are synchronous. There is at most one outstanding system call for any given process (or thread) at any point in time. Sky copies over the system call number, arguments and the PID of the currently executing process while intercepting a system call entry on any of the virtual processors. Sky matches the subsequent system call exit that occurs on a virtual processor with the currently outstanding system call entry on the same virtual processor.

**Split system call entries and exits:** I/O-related system calls that usually involve a disk access often get rescheduled to a different virtual processor after issuing the system call but before the guest operating system returns after processing the system call. Sky

needs to correctly match a system call exit that occurs on the new virtual processor with its system call entry that occurred earlier on a different virtual processor. To this end, whenever a new process is scheduled on a virtual processor, Sky checks if there is an outstanding system call entry in that virtual processor. If so, Sky stores that system call information into a global hash table with the PID or TID as the key. During system call exit on a new virtual processor, the contents of the CR3 register and the SP register are used to identify the process or thread (as detailed in Subsection 3.2.1). In order to match a system call exit to its system call entry, Sky first looks up in this global hash table with the PID or TID of the currently executing process or thread for any matching outstanding system call. If a match is found, it is removed and the system call is processed for insights. A match won't be found if the previous virtual processor is still idle and no new process has been scheduled on it yet by the guest operating system. In this case, Sky looks up each of the other virtual processors for an outstanding system call entry that matches the PID or TID for the system call exit being matched. This look up always succeeds because the outstanding system call entry either has to be in the global hash table or with one of the other virtual processors.

**Handling Signal Handlers:** When there are pending signals for a process, the corresponding userspace signal handlers (if any) are invoked by the guest operating system before a system call exit occurs. The signal handlers could possibly issue new system calls too even before the previous system call is finished. Since the signal handler is invoked using a signal stack, the new system call will have a different userspace SP value during system-call interception. Sky detects such signal-handler invocations by noticing this difference in userspace SP and stores the outstanding system-call information into the global hash table. Subsequently, when the outstanding system call's exit occurs after the signal-handler invocation is complete, Sky looks up the global hash table to find the system call information and processes it for gathering insights.

### 3.3.3 Linux vs. FreeBSD System Call Interface

**Guest operating system identification:** Our prototype takes the guest operating system type as a configuration parameter but it is possible to infer this automatically. Known techniques that use VM memory analysis [118] can be used to distinguish Linux from Windows. Linux and FreeBSD differ in the system call number for `exit` which is the last system call executed by a process and it does not return anything. VM memory analysis coupled with observation of system call numbers, their arguments, return values and frequencies could be used as a general approach to detect the guest operating system type automatically in a future version of Sky. Sky depends on the knowledge about the guest operating system's system calls, their numbers, their parameters and their return codes in order to gather insights correctly. System Calls do not change often between versions of the same operating system but Sky must be able to recognize when such changes happen in order to support that version of the operating system.

Sky handles the following differences between Linux and FreeBSD system call interfaces:

- Thread-related system calls: FreeBSD guest operating system uses system calls like `thr_new`, `thr_kill` and `thr_exit` for threads while Linux guest operating system uses `clone`, `kill` and `exit`.
- FreeBSD `nosys` system call: System-Call numbered 0 in x86-64 FreeBSD is an indirect system call that takes another system call number as its first argument and invokes its system call handler. Sky intercepts such `nosys` system calls and gathers insights corresponding to the actual system call that gets executed.
- For failed system calls, the FreeBSD guest operating system sets the Carry Flag in the virtual processor and returns a positive error code integer while the Linux guest operating system just returns the negated value of the error code integer.

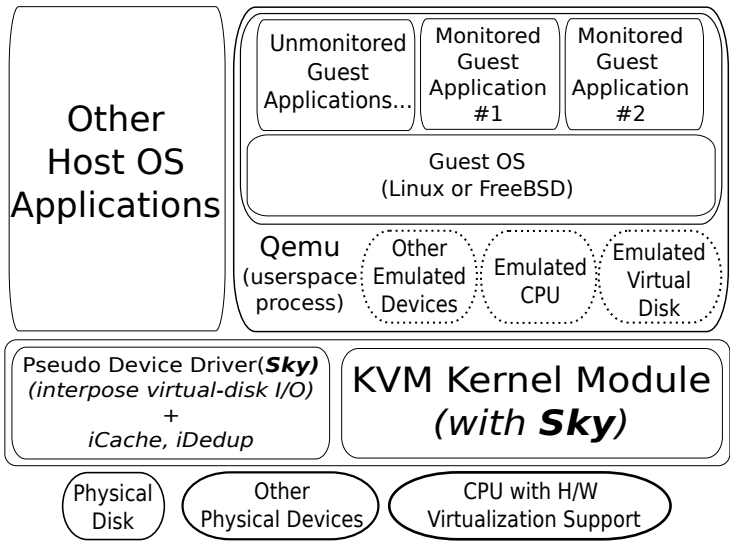



Figure 3.4: **Sky Prototype Organization.** *This figure shows how Sky prototype is implemented for Qemu/KVM as part of the KVM kernel module along with the pseudo device driver within the Linux host operating system kernel. The guest VMs are userspace processes emulated by Qemu.*

### 3.3.4 Prototype

We implemented a prototype of Sky using the KVM/Qemu VMM for the Linux operating system on an x86-64 machine. Figure 3.4 shows the organization of a typical setup of running VMs using KVM/Qemu [27, 109]. The host machine runs a Linux operating system that has a KVM kernel module. Each of the guest VMs is by itself a userspace process running the Qemu emulation program. The KVM kernel module exposes the hardware virtualization features of the processor to accelerate running the userspace Qemu-emulated guest VM.

Almost all of the Sky logic is implemented within the KVM kernel module. We hope that Sky will become part of the mainstream KVM with options to turn it off if users want to. Sky is 7.8 KLOC of new code added to 43.6 KLOC of unmodified KVM source code. This is a modest increase in the hypervisor codebase. Our prototype uses a pseudo device driver in the host operating system (3.8 KLOC of source code) to intercept the I/O requests to the virtual disk rather than intercepting them in the Qemu userspace emulator. This avoids the overheads associated with the communication between host-

| Aspect | Specification |
|---|---|
| Host Processor | Intel i5,3.3Ghz,VT-x/EPT |
| Host OS | Linux (Kernel v3.11.5) |
| Guest OS | Linux or FreeBSD |
| Qemu Version | Qemu v2.5.0 |
| KVM Version | KVM v3.10.1 |
| Host, Guest Memory | 16 GB, 6 GB |
| Virtual Disk | 16 GB Paravirt (RAW Disk format) |
| Backing Disk | 80 GB, 7200 RPM Magnetic Disk |
| Cache Device | 2 GB In-Memory Disk |
| Bcache Version | Comes with Linux Kernel |
| Host FS | Ext3 |

Table 3.3: **Experimental setup.** *This table shows the experimental setup used to evaluate the Sky prototype.*

userspace and host-kernelspace while keeping the number of modified components minimal. Because the I/O requests to the virtual disk are intercepted within the host operating system kernel, Sky will also intercept disk I/O requests from the VMM and the host file system that are necessary for laying out the virtual disk on the backing physical disk. Sky correctly classifies such I/O requests as metadata because they won't be found in the set of data I/O requests tracked by Sky. Sky uses 64 bit checksums calculated using the 64 bit FNV-1a hash algorithm [60]. Bloom filters are used for quick lookups when appropriate: to check whether a process is monitored or unmonitored and to check whether a system call is I/O or process-management related or not. All the results reported in the following sections (Sections 3.4 to 3.8) are the average of three trials.

## 3.4 Overhead Evaluation

Our experimental setup is outlined in Table 3.3. The virtual disk is loaded in KVM/Qemu with cache parameter as "none" so that the Qemu-provided cache is disabled for evaluating the effects of the enhanced caching with insights. For all experiments in this chapter,

the measurements reported as when running without Sky are taken by disabling the Sky relevant code in our modified KVM module as opposed to using an untouched vanilla KVM version. We saw no measured difference in runtime or memory consumed when running a vanilla KVM version versus our modified KVM with Sky related code disabled.

During each VM exit caused by a system-call interception KVM code gets executed in order to figure out the exit reason, to handle the exit and to emulate the instruction that caused the VM exit. In addition to this, Sky performs some computational work and hash table lookups to do the following: check whether the current process is monitored or not, check whether there has been a thread switch since the last system call interception for the same process, check if this is a split system call or if there has been a signal-handler invocation and perform statistics update for timing measurements to aid experimentation. This leads to CPU cache pollution.

We used a set of micro and macro benchmarks to evaluate the overheads due to Sky. The benchmarks were run both with and without Sky. *iDedup* and *iCache* were both disabled for these experiments. We ran these measurements on two different guest operating systems: Linux and FreeBSD. When run without Sky, there is no system-call interception happening. The difference in runtime is used to calculate the overheads introduced by Sky as shown in Table 3.4. The percentage overhead that Sky introduces for real applications and macro benchmarks is minimal (under 5%) as seen from the last five rows of the two Table 3.4. The overall overhead is also split up to show how much of it is due to VM exits, basic Sky system-call interception and insight generation.

**Micro-benchmarks:** We use three types of micro benchmarks to measure the overhead imposed by Sky and their results are presented in Table 3.4. The repeated reads and writes benchmark accesses the same offset in a file leading to no actual virtual-disk I/O. When there is high-latency disk I/O (e.g. Random Writes workload), the overhead

| Workload | With Sky (secs) | Without Sky (secs) | % Total Overhead (Splitup: VM Exits /Sky Interception /Insights) |
|---|---|---|---|
| **With Linux Guest OS and Ext3 FS** | | | |
| Random Reads | 99.4 | 98.5 | 1 (1/ 0/ 0) |
| Random Writes | 54.9 | 54.6 | 1 (1/ 0/ 0) |
| Sequential Reads | 14.6 | 14.9 | -2 (-/ -/ -) |
| Sequential Writes | 25.5 | 23.4 | 9 (4/ 2/ 3) |
| Repeated Reads | 20.3 | 5.7 | 256 (157/41/58) |
| Repeated Writes | 23 | 6.6 | 248 (153/36/59) |
| Encryption | 33.8 | 32.3 | 5 (0/ 0/ 0) |
| File Search | 78.3 | 76.4 | 4 (2/ 2/ 0) |
| File Copy | 24.2 | 24.4 | -1 (-/ -/ -) |
| Mail Server | 385.1 | 381.1 | 1 (0/ 1/ 0) |
| TPC-H (MySQL) | 36.1 | 35 | 3 (3/ 0/ 0) |
| **With FreeBSD Guest OS and UFS FS** | | | |
| Random Reads | 185.2 | 183.3 | 1 (1/ 0/ 0) |
| Random Writes | 272.5 | 269.8 | 1 (1/ 0/ 0) |
| Sequential Reads | 30.1 | 25.5 | 18 (13/2/ 3) |
| Sequential Writes | 50.2 | 39.5 | 27 (24/1/ 2) |
| Repeated Reads | 31.7 | 15.4 | 106 (68/16/22) |
| Repeated Writes | 33 | 15.8 | 109 (68/16/25) |
| Encryption | 25.3 | 24.8 | 2 (0/ 1/ 1) |
| File Search | 54.6 | 53.4 | 2 (1/ 1/ 0) |
| File Copy | 34.9 | 34.7 | 1 (0/ 1/ 0) |
| Mail Server | 163.5 | 160.8 | 2 (0/ 1/ 1) |
| TPC-H (MySQL) | 21 | 21.7 | -3 (-/ -/ -) |

Table 3.4: **System-Call Interception introduced overheads.** *This table compares the time taken for various workloads when run with and without Sky on both Linux and FreeBSD guest operating systems. System-Call Interception was turned on when running with Sky and was turned off when running without Sky.* iCache *and* iDedup *were both disabled. The total percentage overhead is shown and also splitup into sub components of percentage overhead due to VM exits, Sky interception and Sky insight computation.*

introduced by Sky for every I/O request is negligible when compared to the disk latency. However, the overhead of Sky is relatively high compared to the I/O request's latency when there is no disk I/O (e.g. Repeated Write to the same offset of a file). Sky is

designed for I/O applications that issue I/O requests that involve accessing the disk. The repeated reads and writes micro benchmark is a worst case workload for Sky and hence system-call interception should be turned off for such applications.

**Macro-benchmark and applications:** We measured the overhead introduced by Sky for file encryption using the *gpg* command, file search using the *find* command, file copy using the *cp* command, Filebench varmail benchmark and TPC-H query on a MySQL database server. As shown in Table 3.4, the overheads are minimal (under 5%).

**Memory overhead:** Across all the experiments we ran, Sky used a peak memory usage of 33 MB of memory for its data structures including the various hash tables (not shown in Table 3.4). The amount of state maintained by Sky is on the order of 10s of bytes for every 4096 bytes of in-progress I/O (system call has been issued but virtual-disk I/O is not yet complete). Therefore, even for write-intensive workloads with 100s of 4K IOPS and a write delay of 10 seconds due to guest operating system page cache, the memory overhead will be on the order of 10s of MBs.

## 3.5 Case Study #1: Information Gathering

In this case study, we show two examples of information gathering using Sky that are either useful by itself or can be used to improve storage performance.

### 3.5.1 Accuracy of data classification for different file systems

We ran the varmail workload from the Filebench [124] benchmark suite after configuring it to finish after running a total of 100000 operations like file delete, file create, file append, file sync and whole file read. We also modified the Filebench benchmark to report the total number of 4096-byte chunks of data written to files. Sky classifies all

| Guest FS | Guest OS | Misclassification Error |
|---|---|---|
| Ext3,Ext4,XFS, JFS,Nilfs2,Reiserfs | Linux | 0% |
| Btrfs | Linux | 3.9% |
| UFS | BSD | 0% |
| ZFS | BSD | 0.7% |

Table 3.5: **Accuracy of Sky.** *This table shows the data writes misclassification error percentage for Filebench varmail on different file systems.*

the written data with a zero error percentage for all but two copy-on-write file systems as shown in Table 3.5. We saw small inaccuracies for the copy-on-write file systems Btrfs and ZFS (3.9% and 0.7%) which is a limitation of our current prototype. All writes that are not data are classified as metadata. This information is useful to evaluate the accuracy of Sky as well as to take a closer look into performance of different guest file systems as part of a virtualized-storage stack.

### 3.5.2 Block lifetimes

A VMM could use information about block lifetimes in order to schedule write back caching using a persistent cache device, to perform data reorganization on the backing physical disk, to intelligently prefetch content that skips dead blocks or to optimize recovery by skipping dead blocks [161, 186]. Sky allows a VMM to get the block liveness information in a virtual machine setting without modifying the guest operating system, file system or the applications. When the virtual disk and the file system support TRIM or UNMAP commands, Sky could track block liveness with less effort by tracking only those blocks that get deleted and are quickly reallocated before a TRIM or UNMAP command is issued to the virtual disk. Our current prototype of Sky targets applications that can tolerate a small level of inaccuracy due to checksum collisions. A detailed comparison with past related work on block liveness is in Section 5.2.

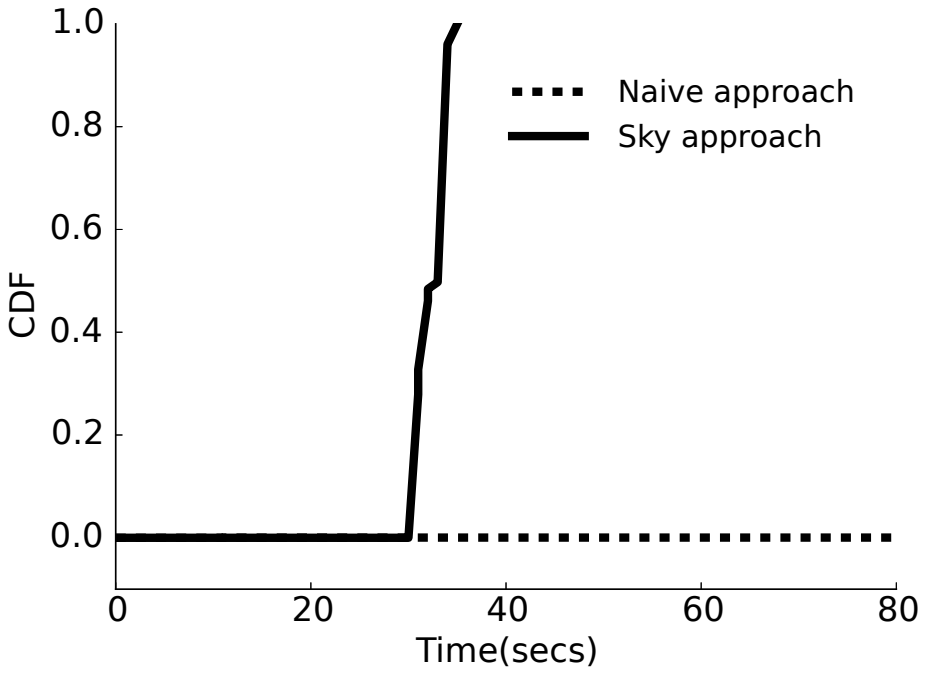



Figure 3.5: **CDF of block lifetimes for a synthetic workload.** *This figure shows the CDF of block lifetimes calculated using two approaches for a synthetic workload that writes 80 MB of data and deletes the files after a 30 secs delay.*

**Approach:** Sky uniquely identifies files by the guest disk device number and inode number. Sky maintains checksums of 4096-byte chunks at various file offsets by intercepting `write` and other related system calls. It detects block lifetimes by detecting overwrites to content already present at various file offsets. Sky also intercepts `unlink`, `truncate` and related system calls to accurately take into account file deletes and truncates. Figure 3.5 compares the block lifetimes calculated using a naive approach that just uses block overwrites with that calculated using Sky for a synthetic workload that writes 80 MB worth of file contents, sleeps for 30 seconds and finally deletes the files. This synthetic workload helps illustrate that Sky correctly handles file deletes. Since the naive approach does not know about the file deletes or truncates, it thinks all the blocks are still alive, while Sky correctly calculates the block lifetimes as approximately 30 seconds for all the blocks.

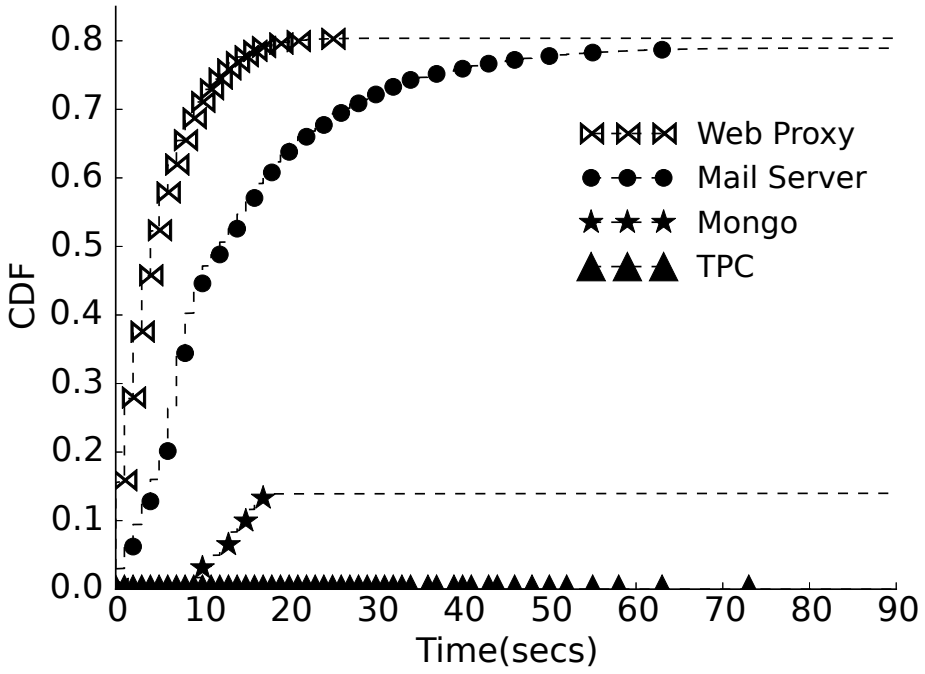



Figure 3.6: **CDF of block lifetimes for Filebench workloads.** *The CDF does not reach 1.0 because there are some blocks still alive at the end of these Filebench workloads.*

Figure 3.6 shows the cumulative distribution function (CDF) of block lifetimes for four Filebench workloads. Sky could use such block lifetime information about the running workloads to adjust the delay while scheduling write-back from a faster persistent cache device to the slower disk. The CDF does not reach 1.0 because there are some blocks still alive at the end of these Filebench workloads.

## 3.6 Case Study #2: *iCache*

In this case study, we show how the effectiveness of a storage cache can be improved for certain workloads by using the policy of caching small files and file system metadata with higher priority. This policy is complementary to the traditional cache-management algorithms like LRU, LFU, MQ [232] and ARC [127]. Such traditional cache-management algorithms differentiate disk blocks only based on their access patterns and do not

| **File Size** ( MB) | **<14** | **<10** | **<5** | **<2** | **<1** | **Meta Data** | **Skip Dedup.** |
|---|---|---|---|---|---|---|---|
| *I/O class* | 0 | 1 | 2 | 3 | 4 | 5 | 32 |

Table 3.6: **Policy to assign I/O class to disk I/O requests.** *The default* I/O class *is 0. Priority increases with increasing* I/O class *values (0-lowest,5-highest). Sky uses* I/O class *32 to hint that the payload is unique and deduplication can be skipped.*

associate any semantic meaning to them. Our work adds this missing semantic meaning to the traditional cache-management algorithms. Sky helps VMs make better use of their fair share of cache allocated by the hypervisor and is complementary to algorithms for fair cache partitioning between VMs. We also show how a MySQL server can be easily modified to pass insights directly to Sky bypassing the guest operating system.

### 3.6.1 Implementation

**Bcache External Disk Caching Module:** We integrated the Bcache block device caching layer from the Linux kernel with Sky's pseudo device driver. I/O to the slower magnetic disk that contains the virtual disk is cached using an in-memory disk. We made the following modifications to the stock Bcache code (adding 10% new code):

- Added statistics collection to track and report sector-level hits and misses rather than the default request-level reporting.
- Added code to search for the slot holding a particular sector and change its priority.
- Added code to not cache guest file system journal writes so that effects of the *iCache* can be compared against the normal cache with more clarity.
- Allowed "clearing the cache" and "reading the list of cached sectors" from the userspace for experimental convenience.

**Enforcing Higher Priority for Metadata and Small File I/O:** Bcache sets the priority of a newly allocated slot to 32K. The priorities of all slots are decremented periodically

based on the amount of data handled by the cache. Upon a read hit, the priorities of the corresponding slots are reset to 32K. The slots with lower priority are reclaimed earlier. We added code to set the priority of slots containing small file data and metadata to higher values between 32K to 64K based on the *I/O class*.

For write requests, the priority of metadata and small file I/O is set appropriately when new slots are allocated in bcache to hold the data because the insight is available by then. However, for read requests, the classification insight will be available only at a later point in time. For data read requests, the insight is known when the corresponding system call's exit is intercepted by Sky and its checksums looked up. For metadata read requests, the insight is known when the remaining checksums are cleared out from global hash table after a configurable sufficiently long delay. Hence, for read requests, the priority of the slots is updated when the insight is available. Data readaheads issued by the guest operating system that are not subsequently read by the guest application before the 4 seconds delay and still remaining in the bcache will have their priorities increased as well thereby avoiding a possible miss if at all the guest application reads them in the future.

### 3.6.2 Evaluation

We now show how the performance of a variety of real applications and macro benchmarks can be improved by using this *iCache*. The policy used by the *iCache* is to give the highest priority to metadata followed by lower priorities for data depending on the size of the file as shown in Table 3.6. The emphasis is on how classification of I/O requests and their differential treatment can bring benefits rather than the particular choice of this policy. Applications that fit a different policy profile can use their own policies as described in Subsection 3.6.2. As an example of such a scenario, we show how a virtual file server in the cloud that serves customers with different priorities can pass down

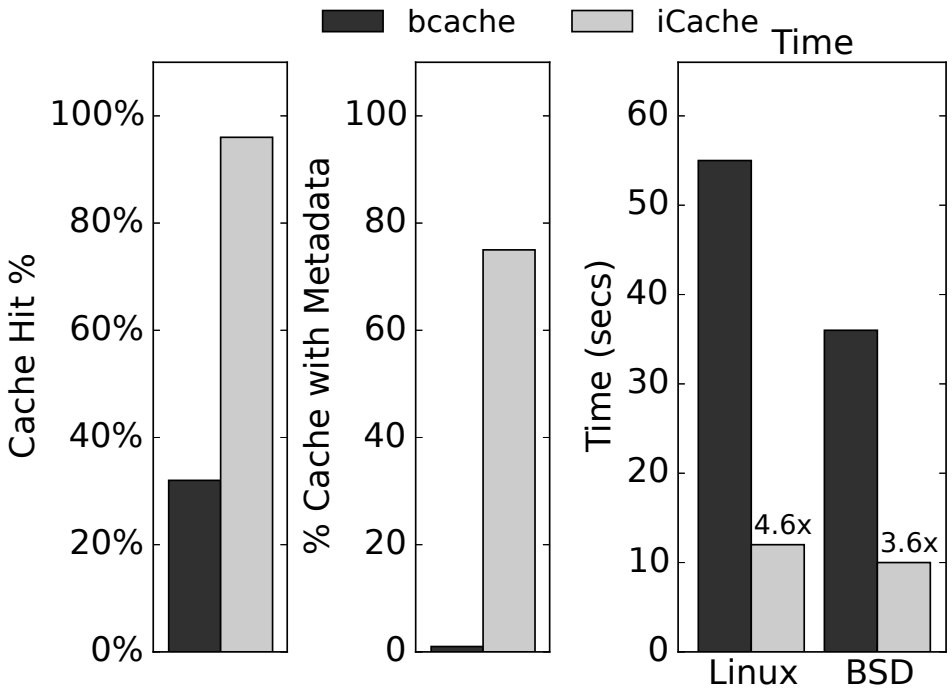



Figure 3.7: **File Name Search (*find*) Results.** *This figure compares the cache hit percentage, cache contents and the runtimes for the file name search workload on bcache and* iCache. *The numbers above the runtime bars for* iCache *are the speedups achieved over bcache.*

this information to the Virtual Disk with very little modification.

We run the experiments both with a normal cache as well as with the *iCache* and compare. The amount of physical resources available is the same for the normal cache and *iCache*. Also, when using the normal cache, system calls are not intercepted, thereby avoiding interception overheads. The differences in results are due to the differential caching done by *iCache*.

**File Name Search (*find*)**

A Linux kernel source code archive of size 115 MB is unzipped and untarred into a newly created file system 10 times creating 450K files with total disk usage of 6 GB. The *find* command is used to search for a non-existent file. The *iCache* retains the file system metadata when the Linux kernel source files are written due to its policy of

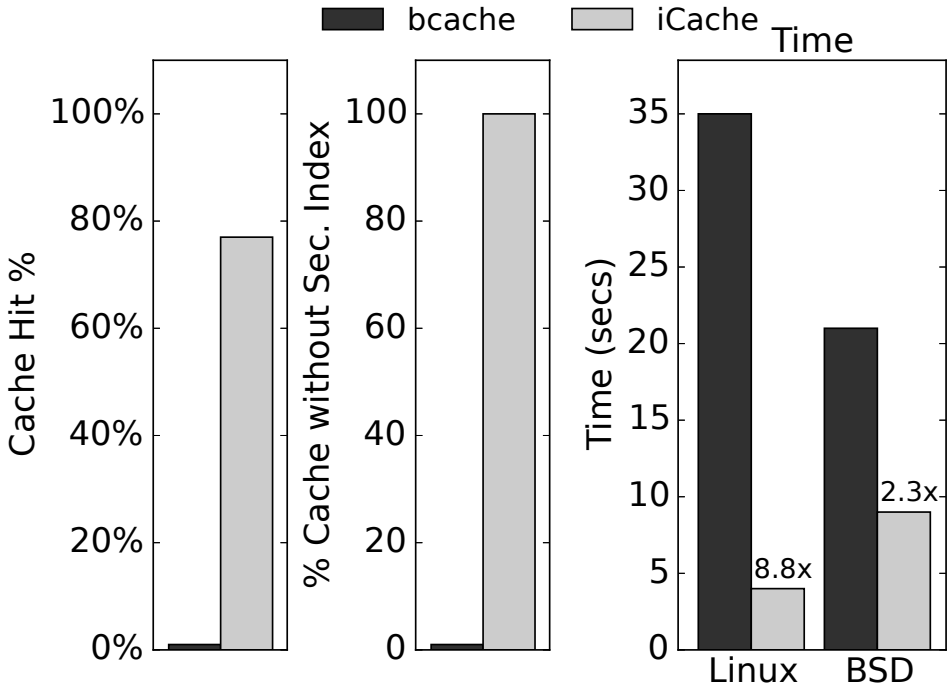



Figure 3.8: **TPCH on MySQL Server Results.** *This figure compares the cache hit percentage, cache contents and the runtimes for a TPC-H query workload on bcache and* iCache*. The numbers above the runtime bars for* iCache *are the speedups achieved over bcache.*

giving higher priority to metadata than for the file contents. Because searching for a file using the *find* command only reads the file system metadata, the *iCache* outperforms the normal cache for this workload by 3.6 to 4.6 times as shown in Figure 3.7.

**TPC-H on MySQL Database Server**

We now show how a more sophisticated real world application can be modified in a way that it can explicitly classify I/O requests for differential caching. We changed the MySQL database server in order to differentiate I/O requests to the Clustered Index (which also contains the table data) from those to the Secondary Index by storing a tag in a thread local store at the various I/O-generating functions. Overheads due to our modifications to MySQL [145] were negligible. The mechanism described earlier in Subsection 3.2.6 is used to pass the I/O class along with the system calls. Since the

secondary index can be huge in size and is also sequential on disk, the current policy we use is to give the secondary index data a lower priority than all other I/O requests. This is similar to the policy used by Mesnier et al. who used filter drivers and kernel changes to modify storage interfaces to pass I/O classification information in Windows and Linux operating systems respectively. They then use the I/O classification information to implement a cache similar to iCache [131]. The advantage of iCache is that it does need modifications to the operating system or the storage interfaces.

We load the TPC-H tables relevant to query number 16 with scale factor 1.7, create a few secondary indices on some columns on the tables and finally execute the query [210]. The *iCache* is able to retain the table data while the secondary index is created due to the policy of lower priority to secondary data. Hence, as shown in Figure 3.8, query number 16 which performs a join on two tables without using any secondary index executes faster on the *iCache* by 2.3 to 8.8 times.

## 3.7 Case Study #3: *iDedup*

In this case study, we demonstrate how the performance of a deduplication system can be improved by using hints about the semantics of the I/O workload gathered by Sky. First, we show how an application that copies one file to another could greatly benefit by avoiding expensive disk-backed hash table lookups. Second, we show how such hash table lookups and additions can be avoided for encryption workloads that very rarely get deduped [198]. The time taken for hash lookups and additions can be substantial because hashes are randomly distributed and it is impractical to keep all the hashes in memory; therefore, a disk-backed hash table that is persisted by frequent flushes leads to slow random I/Os during lookups and additions [122, 233]. When using *iDedup* in real production environments, the hints generation approach could be altered to suit the target environment. For example, our current implementation detects encryption

I/O pattern using application names and destination file name extensions as hints but an alternate approach could flag processes whose I/O requests repeatedly fail to be deduplicated.

### 3.7.1 Implementation

***Dmdedup* Block Layer Deduplication:** We made the following modifications to the stock *dmdedup* [206] code (adding 14.5% new code):

- Added code to specially handle writes that are known beforehand to contain unique payload (e.g. encrypted content) by skipping the initial search and the subsequent addition to the hash table mapping the block checksums to the corresponding physical block numbers.
- Added code to maintain an in-memory cache of block checksum to physical-block number mappings. This in-memory cache is populated during a read issued by a process flagged by Sky as exhibiting the file-copy I/O pattern. *iDedup* checks this in-memory cache for every write before issuing an expensive lookup to the disk backed hash table.
- Added statistics collection to track and report the count of unique and file-copy hints, the hits in the in-memory cache of checksums to physical-block numbers, and the total time spent by all write requests in *dmdedup*.

### 3.7.2 Evaluation

We compare the performance of two different applications on *iDedup* against *dmdedup*. *Dmdedup* uses Dmbufio to buffer the I/O accesses to its disk backend. We ran the following experiments with a Dmbufio size of 1% and 10% of the peak metadata storage needs for the corresponding workload. 1% to 10% metadata cache sizes are typical in real deduplication systems [122]. The system calls are not intercepted for gathering insights when using *dmdedup* avoiding the overheads of system-call interception.

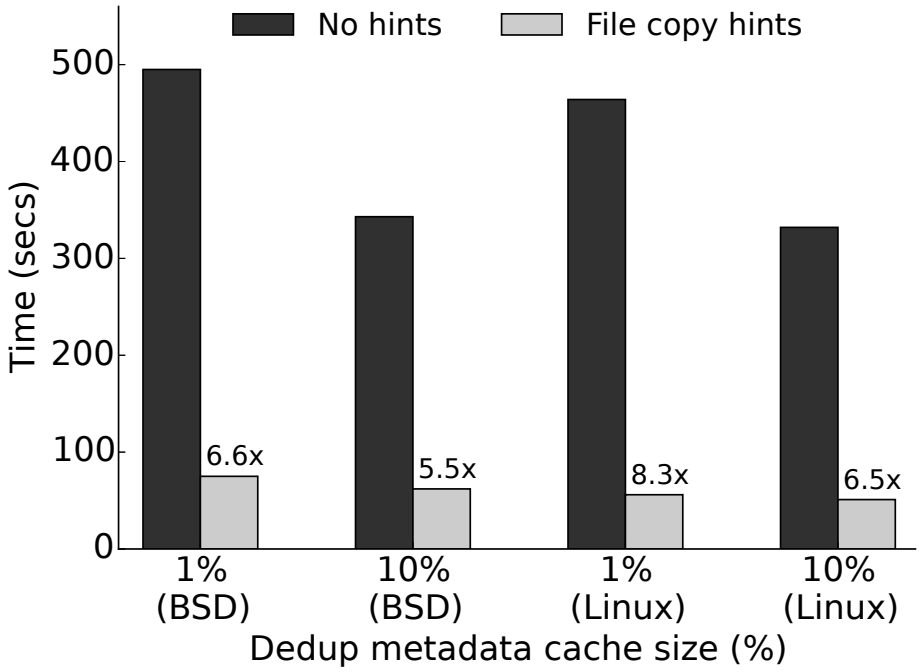



Figure 3.9: **File Copy Results.** *The numbers on top of the light colored bars show the speedup achieved for the file copy workload when run with file-copy hints on* iDedup.

**File Copy (cp)**

The deduplication system is first warmed up by copying a 500 MB file full of random data on a newly created file system. Next the experiment is run which copies the just copied 500 MB file to another new file using the Unix *cp* command. Sky detects the file-copy I/O pattern and sends down hints to *iDedup* for every read issued by file-copy application. For such hinted reads, *iDedup* caches the mapping between the block checksum and the physical-disk block in memory. Upon a subsequent write of the same payload, *iDedup* looks up the in-memory cache and avoids the expensive lookups in the disk-backed hash table. The stock *dmdedup* does not get such hints and hence *iDedup* is faster by 5.5 to 8.3 times as shown in Figure 3.9.

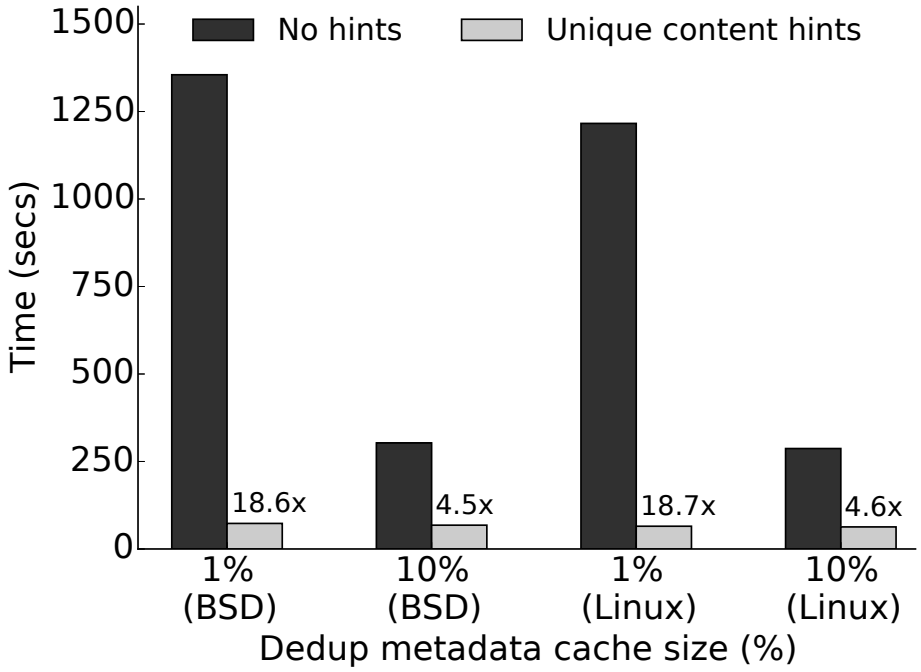



Figure 3.10: **File Encryption Results.** *The numbers on top of the light colored bars show the speedup achieved for the file encryption workload when run with unique content hints on* iDedup.

**File Encryption (gpg)**

A 500 MB file full of random data is encrypted using the GNU Privacy Guard (*gpg*) program. Sky infers the encryption by using the executable name of the *gpg* program and passes down hints to *iDedup* about unique data content. *iDedup* uses the hint to avoid looking up and subsequently adding a new entry to the disk-backed hash table that maps block checksums to their physical-block numbers. Because most file system metadata is unique [122], Sky sends unique hints for all guest file system metadata writes also. *iDedup* is able to improve the runtime by 4.5 to 18.7 times over *dmdedup* as shown in Figure 3.10.

| Case Study | Workload | W Sky (secs) | W/O Sky (secs) | SSD Speedup/ (Overhead) | HDD Speedup |
|---|---|---|---|---|---|
| *iCache* (Section 3.6) | File Name Search | 3.8 | 4.6 | 1.2x | 4.6x |
| | TPC-H on MySQL | 3.9 | 3.6 | (0.9x) | 8.8x |
| *iDedup* (Section 3.7) (with 1% dedup metadata cache size) | File Copy | 32.3 | 34.1 | 1.1x | 8.3x |
| | File Encryption | 31.5 | 468.6 | 14.9x | 18.7x |
| *iDedup* (Section 3.7) (with 10% dedup metadata cache size) | File Copy | 18.1 | 24.8 | 1.4x | 6.5x |
| | File Encryption | 33.5 | 48.2 | 1.4x | 4.6x |

Table 3.7: **Sky with SSD Backing Disk.** *Sky provides good improvements when used with* iDedup *and provides nominal improvements when used with* iCache *on a SSD backing disk. Sky poses about 8% overhead for the TPC-H query on MySQL Server workload alone. The last column shows the speedup achieved on a magnetic disk for comparision.*

## 3.8 Fast Storage Devices

Sky's system-call interception imposes an overhead that is independent of whether a fast or slow storage device is used; therefore, the interception overhead is relatively higher when used with low-latency storage devices. Because of this, the benefits of *iCache* (Section 3.6) on a SSD storage device are not as high as those on a magnetic disk. Table 3.7 lists the speedups achieved with *iCache* and *iDedup* for various workloads on a Linux guest operating system. *iDedup* (Section 3.7) has significant benefits even when used on a SSD storage device (though not as high as when used on a magnetic disk). Decreasing the system-call interception overhead is a good future research direction in order to make Sky more beneficial when used with fast storage devices.

## 3.9 Deployment Scenarios and Considerations

Sky allows gathering insights about guest applications' I/O requests at the hypervisor layer without modifying storage interfaces in virtualized settings. Big companies that run large scale services can use Sky in their data centers in order to gather insights without going through the pains of modifying the guest Operating System and maintaining those modifications with newer versions of the guest Operating System. Infrastructure as a service (IaaS) cloud service providers can also allow their customers to sign up for Sky as a feature that can provide better caching, deduplication and also allow applications to supply hints. Without Sky, the guest Operating System interfaces have to be modified to talk to the hypervisor using a pre-agreed API in order to pass hints about I/O requests.

In this work, we demonstrated the utility of Sky in improving the performance of caching and deduplication systems for a few workloads. Sky is not a generic solution that is able to improve the performance of all types of workloads and applications. For example, the overhead due to system-call interception might not be low when compared to the workload run-times for compute bound applications. Also, for really fast storage devices, the overhead due to system-call interception might not be low when compared to the disk I/O latencies. In the future, Sky can be made to mitigate this using self monitoring. For example, Sky can periodically monitor the performance benefits it delivers for monitored processes (in terms of higher cache hit ratios and deduplication benefits) and then turn-off monitoring those applications which are not benefiting. Our current prototype implementation does not do this yet.

## 3.10 Summary

We first motivate the need for Sky by discussing the benefits of implementing cross-layer optimizations at a single spot – the virtual machine monitor in our case. Pervasive

changes to many layers and interfaces makes it hard to deploy innovations in real systems and subsequently in maintaining them. It is essential for Sky to have information in order to implement optimizations to manage resources efficiently. We present system-call interception and system call injection as a core mechanism that can be used to gain necessary insights about a guest virtual machine and the applications running in it without requiring modifications to the storage layers or their interfaces. We are able to gather the same insights and reap similar benefits as past research work that had to modify certain storage layers and their interfaces.

We then detail the design of Sky and the interception framework it provides. Sky selectively intercepts I/O related system calls from specific guest processes. Sky maintains a lot of detailed information about the processes that it monitors. We also discuss a technique called insight-calls which is used by Sky to read the guest operating system state. We then describe the three insights that Sky gathers for every I/O request: file size information, file system metadata vs. data and application I/O patterns. Sky also allows applications to supply insights as additional arguments to existing system calls.

We then discuss the implementation details for the prototype Sky we developed as an extension to the KVM hypervisor. Our prototype supports both Linux and FreeBSD guest operating systems. We then measure the overhead imposed by Sky for a variety of micro and macro benchmarks. We then perform three case studies using Sky. The first case study allows gathering information about guest applications like: the block lifetimes and the accuracy of guest file system metadata vs. data classification. The second case study implements and evaluates a smart caching system in the hypervisor called *iCache* that gives higher priority to small files and file system metadata. The third case study is about a smart inline deduplication system called *iDedup* in the hypervisor that handles encrypted and file-copy content differently. We finally show that Sky is also effective for the faster solid state drives; though not as much as for the hard disk drives.

# Chapter 4

# Corruption Resilient Check and Repair

With the advent of large scale distributed storage systems that are made up of hundreds and thousands of disks, it is very important to handle disk failures like corruptions. In this chapter, we study the corruption resilience of NoSQL distributed systems and classify I/O requests to selectively enhance corruption resilience and recovery. We selectively store certain files along with replicas and checksums to protect them from corruption. Additionally, we use I/O Classification on file granularity during recovery from corruptions.

A new breed of scalable storage systems plays an increasingly important role in the modern datacenter. Loosely (and occasionally, inaccurately) referred to as "NoSQL" storage, these systems (including Cassandra [113], MongoDB [42], and many others [36, 39, 52, 99, 110]) are designed to be hugely scalable and resilient to faults, while serving a variety of application requirements. Thus it is no surprise to see these systems widely deployed in production; for example, thousand-node Cassandra clusters exist within Apple and Netflix [138].

However, unlike traditional storage, these systems do not yet enjoy the benefits of a rich and well-developed storage management toolchain. For example, features such as cluster-wide, consistent snapshots and wholesale backup/restore are not widely

available [82, 85, 88]. Instead, users rely upon these systems to stay up and working, hoping that the data redundancy within the storage system is enough to keep data available.

One particular area that has received little attention in these storage systems is that of storage checking and recovery. In traditional systems, tools such as `fsck` [29, 125, 126] form a critical part of the overall storage management toolchain. Such tools were essential to recover corruptions resulting from the improper shutdown procedures [126] and other errors. Techniques such as journaling [155, 212] or copy-on-write [31, 80, 159] provide a way for file systems to maintain consistency in the presence of crashes and improper shutdown procedures. But "bad" images still arise due to disk drive failures [10, 14, 14, 23, 48, 56, 66, 70, 92, 102, 112, 133, 140, 146, 153, 156, 174, 175, 178, 179], RAM failures [35, 176, 194] and bugs in software [43, 61, 201, 226]. Software bugs could cause corruptions in three different ways: they can (i) directly alter or corrupt the data being written to disk, or (ii) cause a torn write, wherein only a portion of the data block is written successfully, or (iii) lead to a misdirected write, wherein the data is written to either the wrong disk or the wrong location on disk, thus overwriting and corrupting data [26, 156]. Latent sector errors occur when a sector becomes inaccessible and such errors may occur five times more often than absolute disk failures [20, 173, 177]. Anecdotal evidence has shown the prevalence of storage errors and corruptions [136, 139]. Checker tools are necessary to help a user recover most of their data successfully in such scenarios.

One might think that a sophisticated and complete check-and-recover tool is unnecessary in the era of redundant storage. For instance, each data item is generally replicated and thus a crash of a single node should not lead to data loss; furthermore, the entire system is unlikely to crash and thus leave the system in an incorrect or inconsistent state.

However, we believe that check-and-recovery tools are needed in this environment for the following reasons. First, catastrophic failures are indeed possible; the likelihood of system-wide outages, perhaps leaving the system in a hard-to-recover state, is low but

(unfortunately) non-zero [8, 24, 44, 46, 50, 51, 59, 68, 76, 105, 141, 207, 227]. In such a case, a check-and-repair tool may be needed to fix critical data structures and restart the system. Second, as snapshotting tools become commonplace [104, 168, 181–183, 197, 211], users wishing to access (potentially inconsistent or damaged) snapshots could first make use of a check-and-repair tool, thus enabling them to access the snapshot for further usage. Finally, effective single-node check-and-repair has the potential to speed up local node crash recovery; after a crash, local data structure repair can likely operate more quickly than a full-state restore (via copy from a remote node).

In this chapter, we first analyze existing NoSQL storage systems, in order to better understand their deficiencies and needs with regards to check and repair. Specifically, we perform a thorough study of the corruption-resilience capability of three distributed key-value stores: MongoDB [42], Cassandra [113] and Riak [110]. We empirically find that the check and repair tools that come with these distributed storage systems have poor corruption resilience. For example, in all the three systems we study, the checker tool often leads to poor results (e.g., further corruption) when attempting repair. Specifically, in Cassandra and MongoDB, more than 99.9% of the bytes on disk are not properly recoverable by existing tools.

Based on our analysis, we design and implement DSCK, a distributed storage-system check and repair framework. DSCK includes three major components to enable the construction of a robust check-and-repair tool for a NoSQL storage system. The first component is a local redundant store for critical (and otherwise, non-replicated) meta-data. DSCK intercepts I/O requests at the library-call level and classifies them based on the destination file to enable transparent checksummed local replication only for those specific files for which the NoSQL store does not already maintain checksums. DSCK uses a configuration file with file-path prefixes, patterns and suffixes to achieve the file classification.

Our I/O interception and classification technique is non invasive because it does not

require any modifications to the NoSQL store, the operating system or the file system. Moreover, it does not need super-user or administrator privileges. This makes it easy to adopt DSCK in existing deployments. One alternate invasive approach to achieve the same results would have been to modify the NoSQL store and the system call interface to allow the NoSQL store to explicitly specify to the lower level storage about which files need additional checksums and replication. Another alternate invasive approach would be to just modify the NoSQL store to add checksums for the files that currently don't have checksums. Both these invasive approaches require that these changes be done again in order to support a new NoSQL store. Maintaining backwards compatibility with the past release version of the NoSQL store is also a challenge. This makes it hard to adopt, deploy and maintain such invasive approaches in real deployments.

The second component of DSCK is a generic checker framework that classifies I/O at the file granularity and invokes the checker module for the specific file-type. The checker uses checksums (whether inherent in the system, or added by the interception layer) to determine whether corruption has occurred within the data or metadata of a system. The third component is a recovery tool that classifies the corrupted files based on their file-type and chooses the appropriate recovery strategy. The recovery tool uses the local redundant store as well as a host of other techniques to perform an exhaustive fix of on-disk structures.

We evaluate DSCK by implementing $\text{DSCK}_{\text{Cassandra}}$, a check-and-repair tool that can find and fix problems in a Cassandra storage system. We find that our approach (including I/O interception and additional integrity protection) imposes negligible overhead when adding recoverability to most files stored by Cassandra; additional protection (of the Commit Log) incurs modest overheads under write-heavy workloads. We also show that $\text{DSCK}_{\text{Cassandra}}$ greatly improves the amount of data that can be recovered via repair. Standard Cassandra tools recover only 37.5% of its files and less than 1% of data when corrupted; $\text{DSCK}_{\text{Cassandra}}$ can recover 99% of files and 89% of

data with little performance overhead, and nearly all files and data if some performance loss can be tolerated. Finally, we show that DSCK$_{Cassandra}$ greatly improves local-node restore time, replacing a multi-hour process with a repair that takes minutes, in proportion to the size of the corrupted file rather than the entire node's data.

The rest of the chapter is organized as follows. We first analyze existing systems and their recovery tools (Section 4.1). We then discuss DSCK's design (Section 4.2) and its implementation of the Cassandra check/repair tool (Section 4.3). Finally, we perform various experiments (Section 4.4) and summarize in (Section 4.5).

## 4.1 System Analysis

To better understand the state of the art in modern NoSQL storage systems, we first analyze the behavior of three popular systems – MongoDB, Cassandra, and Riak – that are widely used in deployment [83, 86, 89]. We choose these systems for their importance (each is used in production), source-code diversity (MongoDB, Cassandra, and Riak are written in C++, Java, and Erlang, respectively), and architectural diversity (they each apply different techniques for replica management).

We perform five distinct analyses. First, we perform *corruption resilience analysis* to understand how effective the standard check and repair tools that come with each system is. Second, we enact *file category analysis* to understand what each system stores in the file system and how each item should likely be protected. Third, we introduce *corruption recoverability analysis* to understand the impact of corruption upon files within each system. Fourth, we perform *essential file analysis* to determine which files must be intact during node or system startup. Finally, we find out which files have checksums associated with them for detecting corruption through *corruption detectability analysis*.

Our first analysis unveils weaknesses in existing systems, highlighting problematic aspects of existing tools. The latter four are then critical in identifying how to improve

| Distributed System | Repair Results | Num. Files (%) | Data Size MB (%) |
|---|---|---|---|
| MongoDB (Total 26 Files) | Data Lost (Permanent) | 21 (81%) | 1238 (99.99) |
| | Fails w/ Error | 1 (3.8%) | $7x10^{-5}$ ($6x10^{-6}$) |
| | Finishes w/o Repair | 2 (7.6%) | 0.004 ($3x10^{-4}$) |
| | Repairs | 2 (7.6%) | 0.006 ($4x10^{-4}$) |
| Cassandra (Total 194 Files) | Fails w/ Error | 100 (51.6%) | 16 (2.6) |
| | Infinite Loop | 20 (10.3%) | 528.4 (86.4) |
| | Not Repaired | 2 (1%) | 67.1 (10.9) |
| | Repairs | 72 (37.1%) | 0.14 (0.02) |
| Riak (Total 442 Files) | Not Repaired | 68 (15%) | 0.03 (0.004) |
| | Repairs (Eventually) | 374 (85%) | 853 (99.99) |

Table 4.1: **Corruption resilience of current repair tools.** *This table shows the efficacy of the check and repair tools that come packaged with the distributed systems in repairing corrupted files.*

each system, and thus realize a more robust check-and-recover tool.

### 4.1.1 Corruption Resilience of Current Tools

We now evaluate how effective the standard check and repair tools that come with these distributed storage systems are in detecting and recovering from data corruptions? We empirically find that these tools are not resilient to data corruptions. Cassandra's check and repair tool, called *scrubber*, fails to even run to successful completion when any one of the 61.9% of files it stores on disk is corrupted (Table 4.1). Similarly, MongoDB's tools repair corruption only for two small files.

We perform this empirical evaluation using a temporary Btrfs [159] snapshot of the uncorrupted disk-state for each of the distributed storage system being studied. This uncorrupted snapshot is taken after loading the systems with 500K entries that each consist of an *id* field and 10 other 100 bytes sized fields containing random data using the YCSB [47] benchmark. We corrupt one single file in this temporary uncorrupted snapshot by overwriting it completely with random data or zeroes. Then, we run the repair tool that comes packaged with the distributed storage system and collect its

output. The entire experiment is repeated three times to validate the results. We ignore files containing metrics, configuration settings and diagnostic logs for this experiment because these files are not repaired by the standard check and repair tools. We now discuss the results from this experiment.

**Cassandra:** The standard disk-state check and repair tool in Cassandra is called a *scrubber*. There are two versions of this tool. One called *sstablescrub* that is run offline when the node is down and another called *nodetool scrub* which is run while the node is online [87]. For this experiment, we run both tools one after another on a corrupted disk-state. As shown in Table 4.1, the *scrub* tools (both offline and online) output an Error or throw a *Java Exception* for 51.6% of the files. They get into an infinite loop for 10.3% of the files. The looping arises because the *scrubber* is not able to handle corrupted data when trying to read old SSTables (sorted string tables) and replace them with new ones. The *scrub* tools run successfully without any errors only for the remaining 37.1% of the files that comprise solely of all the SSTable component files that end with "Summary.db", "Digest.crc32", and "TOC.txt". These files are not used by the *scrub* tools for reading the contents of the old SSTables.

**MongoDB:** MongoDB's *repairDatabase* tool aims to be analogous to the *fsck* tool for file-systems but it comes with a few warnings: it must be avoided when there are other replicas or backups available that can be used to restore the corrupted data because it can loose data during recovery. This tool removes the corrupted data but does not recover it [137]. MongoDB allows plugging in a lower-level storage engine like RocksDB or WiredTiger. We configured MongoDB to use RocksDB as the lower-level storage engine. RocksDB is a local key-value store written in C++ that uses log structured merge trees for storing data on disk. MongoDB's *repairDatabase* tool is not properly integrated with the lower-level RocksDB's repair tool and therefore does not invoke RocksDB's repair tool during recovery. Thus, we ran our experiments with a two-step repair procedure: first, the lower-level RocksDB's repair tool is run followed by MongoDB's *repairDatabase*

tool.

The corruption resilience of this repair procedure is shown in Table 4.1. Only two files are repaired after a corruption (that too by RocksDB's repair tool and not by MongoDB's *repairDatabase* tool). MongoDB, when used with the RocksDB storage-engine, stores the bulk of the data in SST (sorted string table) files. Corrupted SST files are archived by RocksDB's repair tool (along with an alert that some data might have been lost). This removes corruption and allows running the *repairDatabase* tool without any errors but data is permanently lost in this process. Alternatively, when we ran the *repairDatabase* tool without first running RocksDB's repair tool, it always crashed with errors like "invariant failed", "out of memory", and "assertion failed" when dealing with corrupt SST files. Since MongoDB does not have *read-repair* or *anti-entropy* features that could recover the lost data over time, this data loss is permanent. Others with extensive experience with MongoDB also make a similar claim: that the MongoDB repair tools are really "corruption-ectomies" because they remove corruption but may not leave behind much clean data and that though these tools are not great, they allow one to get the server running again [42].

**Riak:** Riak has different techniques for repairing different types of files like secondary indices, search indices, LevelDB files, and partitions [84]. We used a custom repair script that first invokes the LevelDB repair, then clears the Anti-Entropy state, invokes a repair on the affected partition and then finally rebuilds the Anti-Entropy state. As shown in Table 4.1, this repair procedure recovers from corruption in almost all files except for 68 critical files that store cluster state, virtual nodes state, and ring status. These files do not have any checksum protection on them. When these files are corrupted, it mostly results in the node failing to boot up or subsequently failing while serving read requests. For the rest of the 374 files that are repaired, the appropriate Anti-Entropy repair procedures must be invoked right away in order to recover the data from the other replicas. The current policy in Riak is to trigger Anti-Entropy check periodically

(when it is enabled). Any data lost during repair will be recovered only during the next scheduled Anti-Entropy check or during a read-repair when the lost data is read by the user.

### 4.1.2 File Category Analysis

The following analyses focus on how data is stored by each system, in order to better understand how to detect and recover from corruptions within each. Each system distributes data and metadata across many nodes within a distributed system, and within each node utilizes a local file system for all such information. Thus, we must first understand how data and metadata are stored within the local file system of each node.

To do so, we perform a *file category analysis*. This empirical evaluation determines what type of content is stored in each file. For example, does a file contain user-supplied data, an index, or configuration information? We automatically classify files into one of ten semantic categories.

Knowledge of file contents is critical for any check-and-recovery tool. For example, a file that contains diagnostic logs need not necessarily be protected from data corruptions using checksums; in contrast, a file containing metadata essential for user-data access must be properly protected to enable detection and repair.

Our analysis automatically categorizes files into one of ten semantic file-types (with occasional manual intervention required). These semantic file-types were created considering how the files differ on aspects like: the information that they contain, the purpose of the information in them, their effect on the distributed storage system when they are corrupted, the possibility of detecting a corruption in them, the possibility of recovery after corruption, the difficulty of recovery, etc. The results of this analysis for the three distributed storage systems is shown under the column titled “File Category” in Figures 4.1 to 4.4.

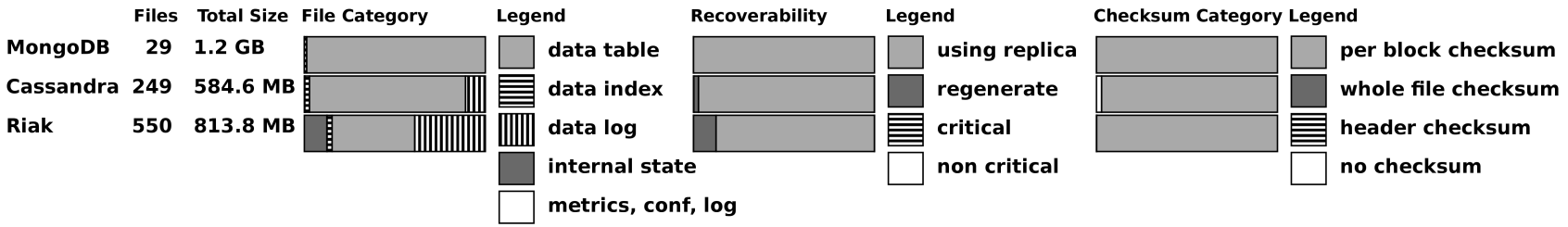


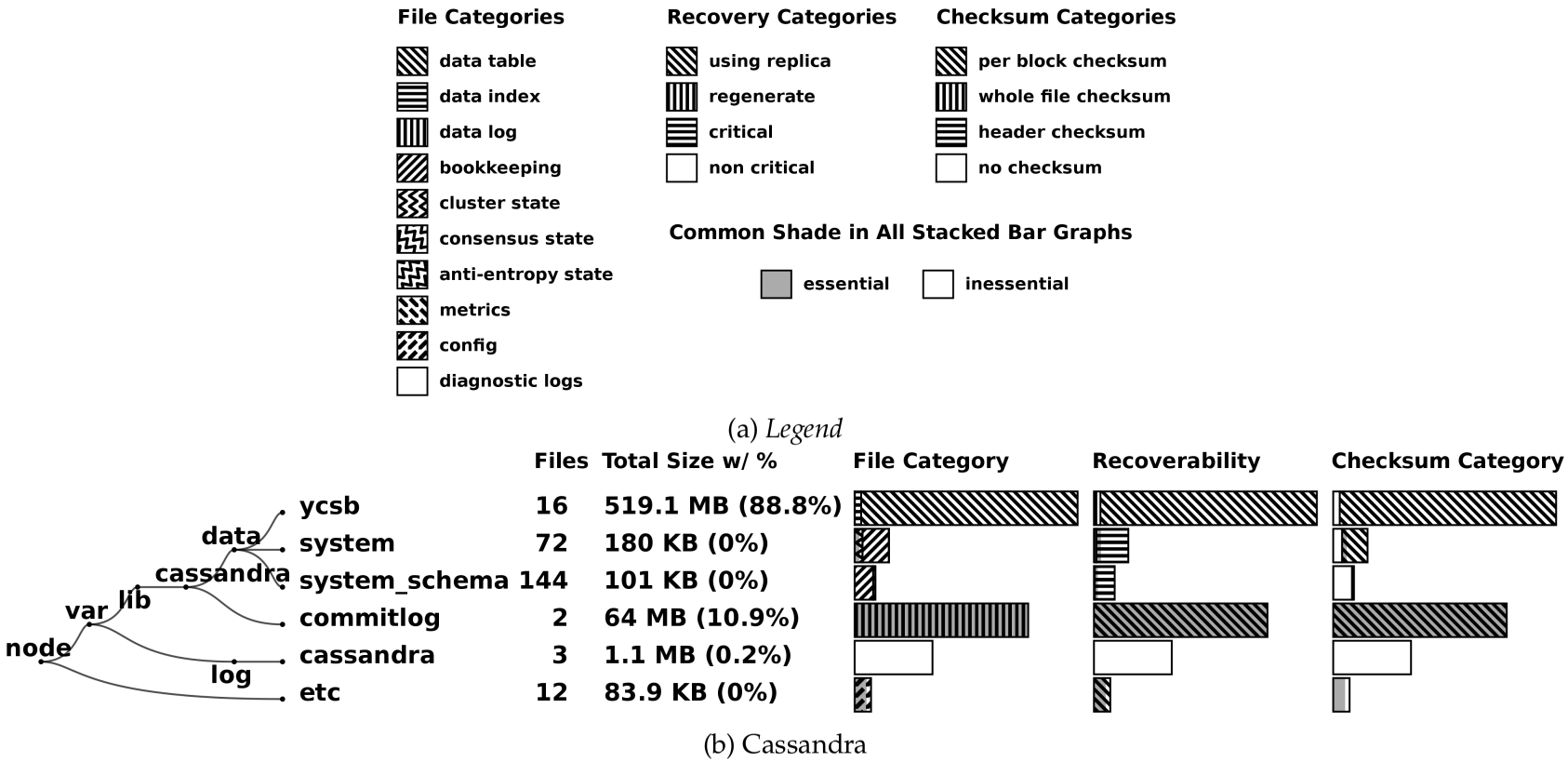

Figure 4.1: **Error detection and recoverability analysis summary.** *This figure shows the amount of data stored on disk by a single node for the YCSB [47] benchmark by MongoDB, Cassandra, and Riak. The on-disk files are categorized by the type of content they contain, the data-integrity protection they have and their recoverability after a corruption.*

The description about the ten different semantic file types follows. The data supplied by the user is stored in three categories of files. The actual table files, categorized as *Data Table*, are the final storage locations of the user-supplied data. Often, the user-data is first written to a write-ahead log for consistent recovery from a crash [170]. Such logs are categorized as *Data Log*. Indices created for the efficient data retrieval from the *Data Table* files are categorized as *Data Index*.

Files containing information about the distributed cluster of nodes such as peer details, ring topology, token ranges, and partition maps are classified under the *Cluster State* category. The *Consensus State* category contains files with information about the consensus reached among the distributed nodes. Files containing information about the divergence of state across peer-nodes with identical data for future reconciliation through anti-entropy protocols are classified as *Anti-Entropy State*. Files categorized as *BookKeeping* contain information like process identities, locks, checksum digests, list of files, identifiers, and manifests. In the cumulative summary graph in Figure 4.1, *Cluster State*, *Consensus State*, *Anti-Entropy State*, and *BookKeeping* categories are represented by a single new category called *Internal State* for brevity.

Files containing debug, error or information logs generated while the system runs are classified as *Diagnostic Logs*. *Config* files contain configuration information used to tune the behavior of the distributed storage systems. Files containing metrics about the various operations that have been completed in the past are categorized as *Metrics*. These three categories are represented as a single new category called *Metrics, Conf, Log* in the cumulative summary graph in Figure 4.1.

As seen in the column titled "File Category" in Figure 4.1, most of the on-disk files stored by the distributed storage system are related to the user-supplied data – they fall under the *Data Table*, *Data Log*, and *Data Index* categories. Compared to other distributed storage systems, Riak maintains more internal state on disk. The major contributor to this is the active anti-entropy state that Riak always maintains about the divergence




(a) *Legend*

(b) Cassandra

Figure 4.2: **Analysis of files stored by Cassandra.** *This figure shows the various files created by Cassandra along with information like: the type of content stored in them, the data-integrity protection available for them and the recoverability after a data corruption. The size of the bar graphs is in log-level scale to improve the visibility of small data sizes.*

of state across peer-nodes. Cassandra, on the other hand, creates anti-entropy state only during a repair session [57]. The total on-disk storage used by MongoDB is almost twice that of Cassandra because all the data inserted using the YCSB benchmark are also stored in the *oplog* collection along with a timestamp for replicating it from the primary to the secondary nodes of a replica set. Old entries in the *oplog* collection has not yet been reclaimed because the total size allocated for it is 990 MB out of which only 596 MB has been used.

Figures 4.2 to 4.4 show how the different categories of files are spread out under the file-system hierarchy for the three distributed storage systems. Files that are of the same type are often stored in the same directory. The directories are named and laid out on disk based on the type of information stored in the files residing in them. This is useful for implementing file-specific check and repair because prefix strings over file names and paths can be used to specify how different file categories need to be protected against corruption, checked for detecting a corruption and recovered from a corruption. For example, prefixes can be used in Cassandra to specify that all files within the *system* and *system_schema* directories are important and they need to be protected against corruption even if the associated costs are high.

### 4.1.3 Corruption Recoverability Analysis

Our next analysis focuses on corruption within the data and metadata of each NoSQL system. Specifically, within this *corruption recoverability analysis*, we classify each file based on the techniques that can be used to recover lost content after a data corruption or inconsistency. To perform this analysis, we both scrutinize source code as well as created automation in order to categorize files into four categories based on: "possibility of recovery", "the ease of recovery", and "the need for new tools for recovery". The results of our analysis are shown under the column titled "Recoverability" in Figures 4.1

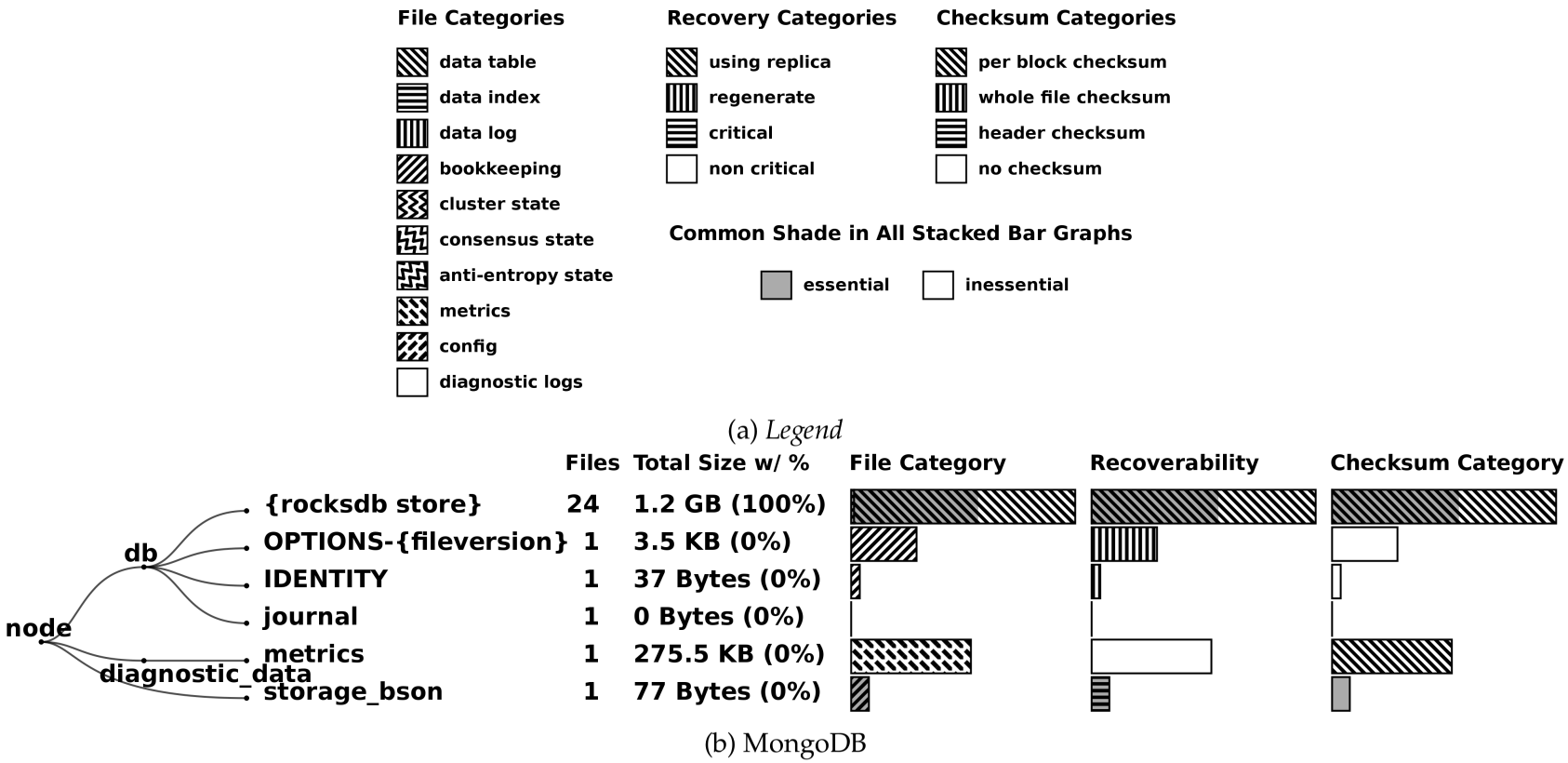
to 4.4. The description about the four categories is as follows.

Certain files are not essential for the correct functioning of the system and hence do not need any recovery technique. For example, files used to log what happened in the system for manual analysis during a future debugging session and files that are useful to know the metrics on the various operations that happened in the past. Such files are categorized as *Non critical*. Files that can be recovered using remote replicas are classified under the *Using Replica* category. Files that do not have any other replica but are important for the health of the distributed storage system are classified as *Critical*. Files that can be recovered by regenerating them using the tools that come with the distributed storage system are classified as *Regenerate*.

### 4.1.4 Essential Files Analysis

We also experimentally evaluate the corruption resilience of each system. We first identify the files that are essential for these systems to start up and call them *essential* files. These files are read by the distributed storage system when it is starting up. When corrupted, these files will prevent the distributed storage system node to start up even after running the standard repair tools that come with the distributed storage system.

Our approach for detecting *essential* files consists of two phases. In the first phase, we bring up a 3-node cluster from scratch, load data using the YCSB [47] benchmark, bring the cluster down, and finally take a snapshot of each local disk using the Btrfs [159] file system. In the second phase, we load the Btrfs snapshot, corrupt one particular file in one of the three nodes by overwriting it completely with random data, run the recovery tools that come with the distributed storage system on the corrupted disk-state, and finally try to bring the cluster up. If the cluster fails to come back up, then the corrupted file is an *essential* file. To speed up this detection process, we automate the process for each of the three distributed storage systems. We confirmed our findings by repeating




(a) *Legend*

(b) MongoDB

Figure 4.3: **Analysis of files stored by MongoDB.** *This figure shows the various files created by MongoDB along with information like: the type of content stored in them, the data-integrity protection available for them and the recoverability after a data corruption. The size of the bar graphs is in log-level scale to improve the visibility of small data sizes.*

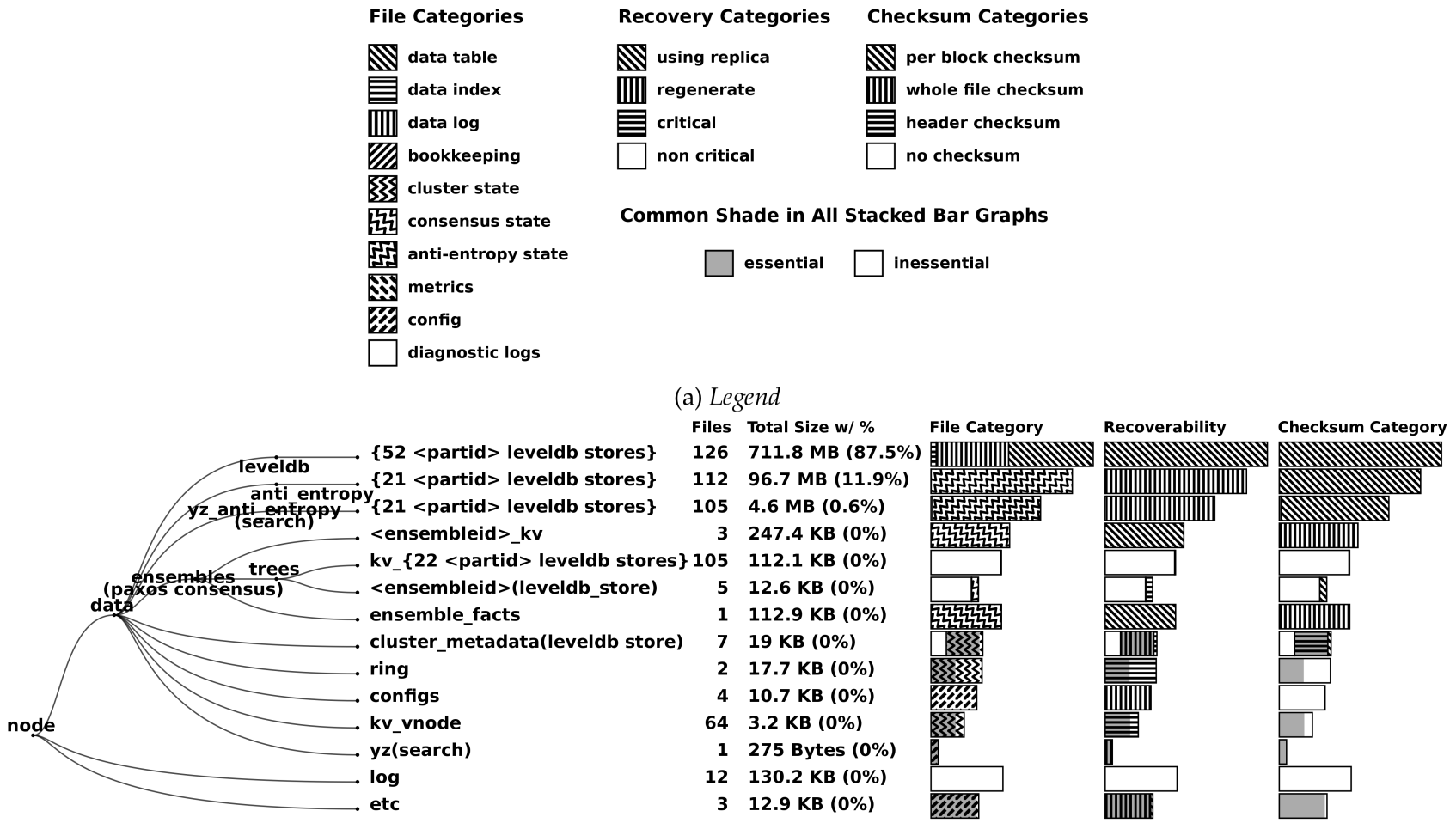
the experiment until confidence was assured.

We correlate the resulting information about these essential files against the other classifications that we do based on "File Category", "Corruption Detectability", and "Corruption Recoverability". This correlation spread out under the file-system hierarchy in these distributed storage systems is shown using a light grey shade in Figures 4.2 to 4.4. As seen in the graphs, files containing *Cluster State*, *Config*, *BookKeeping*, and MongoDB's system-state constitute the *essential files*. These files do not always have checksum protection. Some of them can be recovered using a replica or regenerated while the rest are *Critical* and cannot be recovered easily.

### 4.1.5 Corruption Detectability Analysis

Detecting data corruption is a prerequisite for recovery from the corruption. We now analyze the existing data-integrity protection provided by the three distributed storage systems for the various files they store on disk. This analysis will help us in designing an appropriate solution to make the distributed storage systems completely resilient to data corruptions.

We read through the relevant parts of the source code of the three distributed storage systems to understand the various on-disk file formats they use. We combined this information along with details about well-known file-formats such as BSON and XML to automatically categorize files into one of four levels of checksum-categories that vary in terms of (1) the data-integrity protection they provide over different portions of the file, (2) the computing resources they consume during file access, (3) the complexity of the software needed to maintain the checksums during writes and updates, and (4) the amount of data that needs to be recovered upon a corruption. The results of this *corruption detectability analysis* are shown under the column titled "Checksum Category" in Figures 4.1 to 4.4. The description of the four different categories follows.




(a) *Legend*

(b) Riak

Figure 4.4: **Analysis of files stored by Riak.** *This figure shows the various files created by Riak along with information like: the type of content stored in them, the data-integrity protection available for them and the recoverability after a data corruption. The size of the bar graphs is in log-level scale to improve the visibility of small data sizes.*

Files that do not have any sort of checksum based data-integrity protection for detecting file corruptions are placed under the *No Checksum* category. Files that only have checksums over the header (the first few bytes of the file) are categorized as *Header Checksum*. Corruptions in such files that also modify any part of the checksum-protected header portion can be detected. Files that have complete data-integrity protection are categorized under the *Per-block Checksum* or *Whole-file Checksum* categories. These files have checksums over chunks of data that they contain or over the entire file respectively. Corruptions in any part of the file can be detected. With per-block checksum, the specific chunk that is corrupted can be identified so that recovery can be performed only for the corrupted chunk instead of the entire file.

As seen in the column titled "Checksum Category" in Figure 4.1, most of the on-disk files stored by the three distributed storage systems are protected using checksums. A new corruption-resilient check and repair tool only needs to add checksums for those files that do not already have checksum-protection by the distributed storage system.

### 4.1.6 Observations from the Study

We now make several observations relevant to data corruption detection and recovery in MongoDB, Cassandra, and Riak and discuss their implications for a new corruption-resilient check and repair tool.

***A small amount of on-disk data including important system-state does not have data-integrity protection:*** All the on-disk files that store user-supplied data, which form the bulk of the on-disk state, are protected with checksums by MongoDB, Cassandra, and Riak. However, the auxiliary files generated from the user-data like those that store indices, bloom filters, and statistics are not always protected with checksums. Moreover, other important files that store system-state like table namespaces, list of tables, other metadata, peer-node details, consensus state, configurations, identities, and security

keys are also not always protected with checksums. Corruptions to these important files that store critical system-state can make all the other uncorrupted data in the node inaccessible. For example, in Cassandra, the files that store the statistics and index components of the SSTables (sorted string tables) containing the system-state are not protected with checksums. Even *essential files* that are necessary for the system to start up are not always protected by checksums as shown in Figures 4.2 to 4.4.

Without checksums, it is hard to detect a data corruption and fail-fast. Although traditional file system checkers [29, 125, 126] relied on parsing on-disk state and validating their integrity and sanity in order to detect corruption, many modern file systems use checksums to detect corruptions in metadata, data or both [31, 111, 159, 172, 224]. Therefore, when the distributed storage system reads the corrupted content and interprets it, unknown erratic behavior could occur. It is computationally exhaustive to quantify the ill effects of corruption because it varies based on the specific value of the corrupted content and it also depends on the distributed storage system's code that interprets the corrupted data. For example, when corrupted files containing the bloom filter component of certain Cassandra SSTable files are accessed and interpreted by Cassandra's code, it leads to either "Out of Memory" errors or does not cause any error depending on the specific corrupted value. Cassandra is known to even propagate corrupted data to other correct replicas when compression is turned off [62]. Therefore, when implementing an enhanced corruption-resilient check and repair tool, developer(s) must make sure all the important and critical files stored on disk are protected with checksums.

***MongoDB, when used with RocksDB storage-engine, stores system-state and user-data in the same SST (sorted string table) file on disk:*** It is not a good idea to combine system-state that is critical to the correct functioning of the system along with the user data and store them in a single file because it complicates recovery. For example, Mon-

goDB, when using the RocksDB storage engine, stores cluster state, indices and user-data all into a single RocksDB store. This leads to both system-state and user data getting stored in a single SSTable file on disk. A small corruption to the user-data stored in a SST file that contains both user-data and system-state could lead to the entire SSTable file along with the system-state getting archived by the RocksDB recovery tool. A robust check and repair tool must know such behavior in order to recover corrupted data correctly and completely by leveraging the existing lower-level recovery tools.

***File formats used by some low-level storage engines are not corruption resilient:*** The SST file-format used by low-level data stores like LevelDB, RocksDB and the BitCask file-format used by Riak are not corruption resilient. They lose uncorrupted data while recovering from a corruption. For example, in BitCask data format used by Riak, if there is a corruption in the start of the file and the corresponding index file is also corrupted, then the entire BitCask data file is lost. A new corruption-resilient check and repair tool can perform a complete recovery only if it is aware of all the uncorrupted data that gets lost due to a corruption.

***Fixing a corrupted block in an immutable SST file requires creation of a new SST file with the uncorrupted data:*** The SST file-format used by LevelDB, RocksDB and the SSTable file-format used by Cassandra are not ideal for fixing corrupted blocks. The easiest way is to create a new SSTable containing the uncorrupted data. Ad hoc solutions that do not require a full data copy are possible in certain cases but they are not guaranteed to work with future versions of distributed system software. A new corruption-resilient check and repair tool will have to delete the corrupted SSTables and regenerate them by using techniques like read-repair and anti-entropy or by explicitly fetching the relevant content from a remote replica.

***Recovery tools in some low-level storage engines are not thorough and they are not integrated with the repair tools in the high-level distributed storage system:*** The recovery tools in RocksDB and LevelDB archive the entire SST (sorted string table) file when a single block of data is corrupted. The tools output that some data could be lost because of archiving but do not explicitly say what data has been lost. Moreover, these low-level storage engines are not integrated with the higher-level repair tools. For example, MongoDB's *repairDatabase* tool does not invoke RocksDB's repair tool. In such scenarios, a system administrator needs to know enough details about the internals of the distributed system to figure out what data has been lost and how to recover the lost data. Alternatively, a full repair, if available, needs to be run which could consume significant time and resources. A new corruption-resilient check and repair tool that leverages the existing recovery tools must be wary of this behavior to implement a correct and complete recovery.

***Certain corrupted files can be regenerated by just deleting them and triggering their regeneration:*** There are some files that, when corrupted, lead to incorrect behavior of the distributed storage system but do not cause harm if they are deleted and regenerated. Some such files are regenerated by the distributed storage system software automatically after detecting their absence. While others can be regenerated by invoking certain tools. For example, in Riak: a file named "riak_core_ring.default.20161018190730" that contains information about the ring containing the distributed nodes is successfully regenerated after a corruption by just deleting it and then restarting the node. In Cassandra, auxiliary *Index.db* components in a SSTable can be deleted after a corruption and the *scrubber* tool can be executed to regenerate them. However, when these files are left corrupted, the distributed storage system fails to boot up even after running the check and repair tools that come with the distributed storage system. A new corruption-resilient check and repair tool can take advantage of this observation by just deleting

and regenerating corrupted files instead of repairing them.

### 4.1.7 Summary

Most of the user-stored data is protected using checksums while some small files that are critical for accessing the rest of the data are not protected with checksums. An unrecoverable corruption to these critical files could make all other data inaccessible. Some critical files can be regenerated after a corruption by just deleting them and triggering their regeneration using existing tools. The repair procedure in all the three systems do not use the remote replicas to immediately recover corrupted content that is also available in a remote replica; Cassandra and Riak rely on anti-entropy to eventually recover the lost data while MongoDB permanently loses data.

## 4.2 DSCK

DSCK is a framework that can be used to implement file-specific corruption detection and recovery. It provides a corruption-resilient store that can be used to store important files that are hard to recover after a corruption. Mirrored copies of such files are kept along with checksums. DSCK allows writing customized plug-in scripts that can check and recover specific file categories. The goal of DSCK's check and repair tool is to be able to detect and recover from data corruption in files. DSCK works on the offline disk-state when the node is not running. We expect DSCK to be invoked by the user, the system administrator or through a watchdog daemon that notices that the NoSQL application has crashed. DSCK does not attempt repairs of file-system metadata. DSCK avoids the need to fall back on the time-consuming recovery option of replacing the entire node.

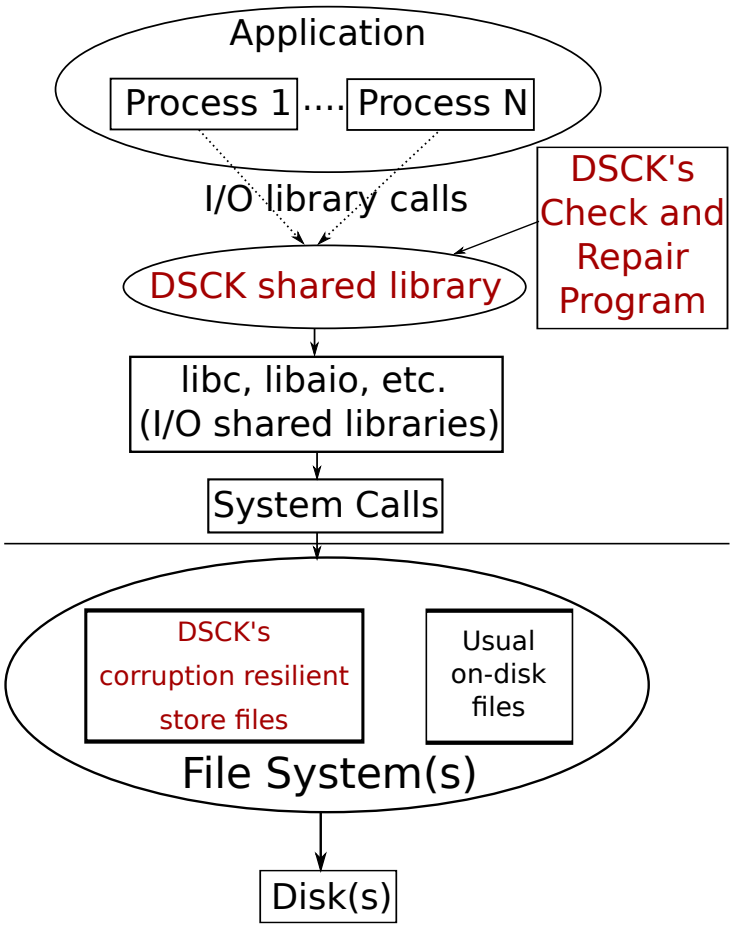



Figure 4.5: **DSCK components.** *This figure shows the components of DSCK: the shared library used to maintain the corruption-resilient store, the checker and the repairer placed alongside the other components of the distributed storage system and the operating system.*

### 4.2.1 Design

DSCK comprises of three main components. Figure 4.5 shows the various components of DSCK and where they are placed with respect to the distributed storage system.

**Corruption-Resilient Store:** This component transparently stores a selected set of files, that do not have any data-integrity protection provided by the distributed storage system, in a corruption-resilient fashion by maintaining locally mirrored copies of the files along with checksums over 4 KB chunks. The mirrored copies and the corresponding checksum files are stored on disk at a location that is spatially distant from the original file that is being protected. The corruption-resilient store makes the task of writing checkers easier

because the checksums can be used to verify a file's integrity instead of parsing the file contents to validate by checking for sane values.

**Checker:** This component uses the checksums to detect any data corruptions on the offline disk-state of the distributed storage system.

**Repairer:** DSCK allows custom scripts to be plugged in that can be used to check and recover files of a specific file-type in a given distributed storage system. Often, existing check and repair tools that come with the distributed storage systems can be leveraged in writing such scripts. The recovery tool uses the techniques described elsewhere (Subsection 4.2.4) to recover any corrupted files.

### 4.2.2 Corruption-Resilient Store

The corruption-resilient store maintains local replicas of a selected set of files along with 32-bit CRC checksums over 4 KB blocks to detect corruptions to the files stored in the corruption-resilient store. Only those files that do not (originally) have data-integrity protection within the distributed storage system are chosen for storage in the corruption-resilient store. Ideal candidates for corruption resilience through this store are: files that are critical to the health of the system but have a complex format that makes it challenging to write robust checkers and validators. DSCK places the local replicas on a separate disk or in regions of the same disk that are spatially distant from each other as well as the protected file because of the spatial locality exhibited by latent sector errors [19, 173]. Our current implementation of DSCK uses a modified *libjio* [28] library to atomically update the corruption-resilient store.

#### Selective I/O Interception

MongoDB, Cassandra, and Riak maintain application-level checksums for user-supplied data which occupies the bulk of the disk space as shown in Figures 4.2 to 4.4. However,

there is no data-integrity protection for some files that are critical to access the rest of the data. For example, Cassandra does not have data-integrity protection for the non-data files within an SSTable. Our goal is to complement the pre-existing partial data-integrity protection provided by the distributed storage system. DSCK does not add checksums for those files that already have data-integrity protection provided by the distributed storage system and for those files that can be easily recovered after a corruption. DSCK uses a configuration file for each distributed storage system that dictates which files need corruption resilience through DSCK by using patterns, prefixes, and suffixes over file names and paths. We evaluate DSCK's overhead with different configurations for Cassandra in Section 4.4. These configurations differ in the overheads they impose (Table 4.5), the corruption-resilience improvement they provide and the corruption recovery time.

All three distributed storage systems studied use standard libraries like *libc* and *libaio* to interact with the operating system. We intercept the I/O related standard library calls by using a shim shared-library that is preloaded before the standard libraries. The library calls made by the application invoke the shim library which acts a proxy to the standard library calls. Using the shim library, DSCK transparently maintains checksum-protected local replicas of a selected set of files by intercepting `write` related library calls. More details about how the I/O interception shim library is implemented can be found later (Section 4.3). This approach is minimally invasive making it easy to adopt and deploy. It does not require superuser privileges and it works with any file system chosen by the system administrator and does not require modifications to the distributed storage system, runtime engines, libraries or the operating system. Moreover, adding checksums at a higher level in the storage stack instead of a lower level provides better data-integrity protection [167, 230]. Although we have not ported DSCK to different operating systems, this technique of using a preloaded shim library is known to be portable across Linux, Windows, and Solaris [123].

**Maintaining I/O Related State**

DSCK maintains certain I/O-related information about each process in per-process map (from C++ standard library) data structures. Information is maintained only for those files for which DSCK selectively intercepts the I/O as described in the previous section. A map called *fdinfo*, with the open file-descriptor number as key, contains information about open file descriptors like: the corresponding file path, the device id, the inode number, whether it is a directory file or a normal file, whether it is opened in synchronous mode and a read-only file descriptor if the file was opened in write-only mode. A second map called *filedirtyinfo*, with the device id and the inode number as the key, contains the regions of the file that were dirtied since the last time the on-disk state was synchronized for this file using library calls like `fsync` and `fdatasync`. A third map called *mmapinfo*, with the memory-mapped regions of the address space as key, contains information including the original application specified memory protection during the `mmap` library call, the backing file description, the offset in the file at which the mapping starts and the set of pages that have been dirtied since the last time the on-disk state was synchronized using the `msync` library call.

**Maintaining Locally Mirrored File Copies**

DSCK intercepts write related library-calls like `write`, `pwrite`, `pwritev`, etc., and performs different actions based on whether the file was opened in synchronous mode or asynchronous mode. If the file was opened in synchronous mode, then DSCK atomically updates the corruption-resilient store with the write before returning control back to the application. If the file was opened in asynchronous mode, then DSCK updates the *filedirtyinfo* map with the regions of the file that are being modified. DSCK also intercepts the library calls that synchronize the on-disk state like `fsync` and `fdatasync` to update the corruption-resilient store atomically with the changes to the file since the

last synchronization of the on-disk state.

**Handling Memory-mapped I/O**

DSCK intercepts the initial `mmap` library call and checks if the request is backed by a file for which a corruption-resilient copy must be maintained as dictated by the configuration file. It then checks if the file is writable and if this is not a read-only memory map request. If all these checks evaluate positive, then DSCK write protects the memory-mapped address space region. Moreover, during initialization at the process-creation time, the DSCK shim library registers its own *Segmentation-Fault handler* in order to trap segmentation faults.

Any write accesses to the memory-mapped address space region will lead to a segmentation fault causing DSCK's segmentation fault handler to be invoked. Upon intercepting a write access to a page, DSCK notes down that the region of the file address space has been modified in the *mmapinfo* map before removing the write protection on the page. Subsequently, when an `msync` library-call is intercepted, DSCK updates the corruption-resilient storage atomically with the modifications since the last synchronization of the on-disk state. DSCK also write protects the associated pages before synchronizing the modifications in order to detect future writes to these pages. The DSCK installed segmentation-fault handler forwards the fault to the original fault handler pre-installed before itself in case the faulting address is not a memory-mapped address that DSCK monitors.

**Ensuring Crash Consistency**

After a crash, DSCK first uses the recovery function provided by *libjio* [28] library to bring the files that were being modified during the crash to a consistent state. Then, DSCK ensures that the special files and their replicas in the corruption-resilient store match. This step is necessary because after a crash the special files could contain writes

that have not yet been synchronized by the distributed storage system using library calls like `fsync`. In this scenario, their versions in the corruption-resilient store will not contain these writes. The distributed storage system does not expect these writes to be persisted to the disk because it has not yet called any synchronizing library calls like `fsync`. We want to prevent the distributed storage system from reading the writes that are not yet in the corruption-resilient store, because it could lead to divergence between the data stored in the corruption-resilient store and the actual files stored by the distributed storage system. Ensuring that the distributed storage system starts with an on-disk state that has the same content as the corruption-resilient store avoids this problem.

### 4.2.3 Checker

The checker component of DSCK uses the checksums that are provided by the distributed storage system and the checksums it additionally maintains in the corruption-resilient storage to detect corruptions. DSCK leverages any checker tools that come as part of the distributed storage system to detect corruptions using the checksums that were added by the distributed storage system. DSCK detects any corruptions to files stored in the corruption-resilient storage using the checksums that it additionally maintains.

### 4.2.4 Repairer

We will now discuss the various recovery techniques that DSCK uses to recover the lost data once a corruption has been detected.

*From DSCK's Corruption-Resilient Store:* If a local corruption-free mirrored copy of the file is available in the corruption-resilient store, then the file is recovered using this copy.

*From a Remote Replica:* DSCK allows using recovery scripts that can recover data lost due to data corruptions using a remote replica when possible. These scripts use any repair

tools that already come with the distributed storage system. The advantage of these recovery scripts is that they invoke the appropriate recovery tools automatically based on what data is corrupted. Without the DSCK recovery scripts, the repair tools that come with the distributed storage systems need to be invoked manually by the system administrator appropriately depending on the data that was corrupted. Alternatively, a full repair needs to be run which could consume significant time and resources.

*Using Software Regeneration:* We observed that, for some corrupted files, just deleting them and triggering their regeneration suffices. For example, an easy way to recover a corrupted immutable Cassandra SSTable is to just delete all the files for that SSTable and then invoke the *nodetool repair* command for the *keyspace* and *tablename* that contained the corrupted SSTable. However, this technique should be used with caution because regeneration could be time consuming for some files.

*Node Replacement:* In the unlikely scenario when an alternate ideal faster recovery option does not exist, DSCK falls back to the time-consuming straightforward recovery technique of taking out this node and replacing it with a new node. However, $\text{DSCK}_{\text{Cassandra}}$, a corruption-resilient check and repair tool we built using DSCK for Cassandra, does not yet use node replacement for recovery.

## 4.3 Implementation

We now discuss two critical aspects of our implementation. The first is a generic I/O interception mechanism; any check-and-recover tool built with DSCK could utilize this library as needed. The second is the detailed implementation of $\text{DSCK}_{\text{Cassandra}}$, a thorough check-and-recover tool for the Cassandra storage system.

### 4.3.1 I/O Interception Mechanism

DSCK uses a preloaded shim shared-library that has proxy functions for the standard I/O-related library calls in libraries like *libc* and *libaio*. The DSCK shared-library is written in C++ and is approximately 3800 LOC. Proxy functions for multiple libraries like *libc* and *libaio* can coexist in a single shim library. Listing 4.1 shows the pseudo-code of a typical DSCK proxy function.

### 4.3.2 Check and Repair for Cassandra

We now discuss implementation details of the corruption-resilient check and repair tool we built for Cassandra using DSCK known as $\text{DSCK}_{\texttt{Cassandra}}$. We first discuss which data we must protect, then how the corruption-resilient store is utilized, and finally how the check and repair tools operate.

**Cassandra Data:** The bulk of the data stored by a Cassandra node falls under the */node/var/lib/cassandra/data* directory which contains SSTables that each store a variety of information including *system-state* and *user-supplied data*. The SSTables containing the *system-state* are small in size, on the order of a few hundred KBs. SSTables, by design, are immutable and are written once but read many times. Each SSTable is composed of 8 different files. A compressed SSTable is made up of 8 files with a common prefix like *mb-1-big-* and the following suffixes: *CompressionInfo.db*, *Data.db*, *Digest.crc32*, *Filter.db*, *Index.db*, *Statistics.db*, *Summary.db*, and *TOC.txt*. An uncompressed SSTable contains a file ending with *CRC.db* instead of the file ending with *CompressionInfo.db*. Of these 8 files, the file ending with *Data.db* contains the actual user-supplied data and is checksummed by Cassandra. The other 7 auxiliary files, including the files that end with *CRC.db* and *CompressionInfo.db* that are critical to access the checksums, are not themselves protected with checksums.

**Corruption-Resilient Store:** The default policy used by $\text{DSCK}_{\texttt{Cassandra}}$ is to store all

Listing 4.1: Pseudo-code for a typical proxy function in DSCK's shim library.

```
ret_type fn_name(arg1,arg2,..,argN){
  //set real_fn_name = original lib func
  if(unlikely(!lib_initialized())) initialize_lib();
  //pre-process this intercepted call
  pre_process_fn_name(arg1,arg2,..,argN);
  //call the original library function
  ret_type res=real_fn_name(arg1,arg2,..,argN);
  //post-process this intercepted call
  post_process_fn_name(res,arg1,arg2,..,argN);
  return res;
}
```

| Criteria | Configuration | Values |
|---|---|---|
| Exclusion | File Name Prefix | "mb_txn_flush", "mb_txn_compaction", "-CommitLog" |
| | File Path Contains | "/hints/", "/saved_caches" |
| | File Name Suffix | "-Data.db", "-Index.db" |
| Forced Inclusion | File Path Contains | "/data/system" |

Table 4.2: **DSCK$_{\text{Cassandra}}$ default configuration.** *This table shows the files that are protected using the corruption-resilient store with the default configuration used by DSCK$_{\text{Cassandra}}$. Files that match the "Forced Inclusion" criteria are protected even if they match an "Exclusion" criteria.*

files except the following in the corruption-resilient store: the *CommitLog*, the temporary logs maintained during SSTable creation or compaction, the cache and hints files that are used for performance enhancement and the files ending with *Data.db* and *Index.db* containing user-supplied data. However, DSCK$_{\text{Cassandra}}$ maintains transparent corruption-resilient copies of files ending with *Data.db* and *Index.db* that contain system-state. Table 4.2 shows this default configuration. The Cassandra *CommitLog* files are temporary log files that are 32 MB in size each and are created periodically. These files are read from only during recovery after a crash.

An alternate policy that includes files skipped by this default policy is possible but will incur additional overheads. The overheads imposed by a few such alternate policies are shown in Table 4.5. Apart from the default policy, we evaluate three more policies

that additionally protect the following files using DSCK's corruption-resilient store: *CommitLog* files, *Index.db* component files for SSTables storing user-supplied data, and both. Each of these policies protects a different set of files using DSCK's corruption-resilient store and therefore differ in the overheads they impose, the corruption resilience they provide and the corruption recovery time.

**Checker:** The $\text{DSCK}_{\text{Cassandra}}$ checker, when invoked over the on-disk state of a Cassandra node, first checks for any data corruptions. $\text{DSCK}_{\text{Cassandra}}$ uses the Cassandra created checksums to verify the integrity of files ending with *Data.db* that contain user-supplied data. For the files ending with *Index.db* that contain indices over user-supplied data, the checker verifies their integrity by using the knowledge about the format of the *Index.db* file and *Data.db* file. An inconsistency at this stage indicates a corruption of the *Index.db* file because the integrity of the *Data.db* file is verified in the previous step using Cassandra created checksums. For the remaining files that are protected by DSCK using the corruption-resilient store, $\text{DSCK}_{\text{Cassandra}}$ uses the checksums that it maintains to verify their integrity. Corruptions in the hints and cache files that are used by Cassandra for performance enhancement are neither checked nor recovered but are simply deleted during recovery. We wrote the core checker logic for Cassandra in the Java language in 451 LOC by reusing the classes that come with the Cassandra source code.

**Repairer:** $\text{DSCK}_{\text{Cassandra}}$ uses the following techniques to bring the on-disk state to a corruption-free state: the corruption-resilient store is used to recover corrupt files that are protected by it, a modified version of the *nodetool scrub* tool that comes with Cassandra is used to regenerate the corrupted *Index.db* files from the corresponding *Data.db* files, the remote replicas are used to recover corrupted *Data.db* files containing user-supplied data by first deleting the SSTables containing them and then using either the anti-entropy [57] (with the *nodetool repair* command) or the read-repair [58] feature in Cassandra. We wrote part of our recovery tools in the Java language in 556 LOC. We also use shell scripts to invoke the core checker logic and then the appropriate recovery

| Aspect | Specification |
|---|---|
| Memory | 128 GB |
| Processor | 2x Intel E5-2630 2.4GHz (16 cores) |
| Disk Drive | 1.2 TB HDD and 480 GB SSD |
| Network | Onboard 1Gb |
| OS, FS | Linux (Kernel 4.4.0), Ext4 |
| Software | MongoDB-3.2.9, Cassandra-3.11.0, Riak-2.0.2 |

Table 4.3: **Experimental setup.** *This table shows the setup used for our experiments.*

| Workload | Without DSCK (secs) | With DSCK (secs) | % Runtime Overhead | % DSCK Added Writes | % DSCK Added Reads |
|---|---|---|---|---|---|
| Seq. Reads | 2.13 | 2.09 | -1.88 | - | - |
| Seq. Writes | 2.27 | 6.82 | 200 | 201 | - |
| Seq. RW Mix | 2.11 | 2.89 | 37 | 201 | - |
| Rand. Reads | 3.42 | 3.43 | 0.29 | - | - |
| Rand. Writes | 2.27 | 23.9 | 952 | 815 | - |
| Rand. RW Mix | 3.4 | 12.34 | 263 | 425 | 28 |

Table 4.4: **DSCK$_{\texttt{Cassandra}}$ overheads for micro-benchmarks.** *This table shows the run time of flexible I/O tester (fio) tool for various micro-benchmarks when run with and without DSCK.*

| Setup | Runtime (secs) | Overhead |
|---|---|---|
| Standard | 530.4 | – |
| DSCK (default) | 533.1 | 0.5% |
| DSCK (+index) | 662.5 | 24.9% |
| DSCK (+log) | 771.3 | 45.4% |
| DSCK (+index,log) | 774.2 | 46% |

Table 4.5: **DSCK$_{\texttt{Cassandra}}$ overheads.** *The table shows the run time of Cassandra when using varying levels of protection using the corruption-resilient store.*

techniques depending on the corruption. These shell scripts total 312 LOC.

## 4.4 Evaluation

We answer three questions in the evaluation of DSCK$_{\texttt{Cassandra}}$. First, what is the cost of using it during runtime, under a write-heavy workload? Second, how effective is it as

compared to standard Cassandra repair tools? Finally, how much more quickly can we repair a node rather than reinstall it from scratch?

### 4.4.1 DSCK's Overhead

In the following experiments, we measure the costs involved in using DSCK to maintain checksummed replicas by using micro-benchmarks as well as the Cassandra NoSQL store.

**Micro-benchmarks:** We first measure the overhead imposed by DSCK on various micro-benchmarks using the flexible I/O tester (fio) benchmark tool [17]. All these workloads work on 20 different files each of size 50 MB amounting to a total of 1 GB. For the read-write mix workloads, we use a 60-40 configuration that issues 60% reads and 40% writes. All the I/O requests issued are synchronized to the disk using an fsync at the end of the workload. We configured DSCK to use a separate SSD disk other than the SSD disk used by the workload. As seen in Table 4.4, DSCK does not add any overheads for read only workloads. For the sequential writes and the sequential read-write mix workloads, DSCK introduces 2x the total number of writes to maintain checksummed replicas. This is because DSCK first duplicates the dirty data to a journal file and then checkpoints them to the actual replica file. The additional work done by DSCK for the sequential writes and the sequential read-write mix workloads leads to 200% and 37% overhead in the runtime. Our current implementation of DSCK handles random writes, when maintaining checksummed replicas, by reading chunks of sequential file regions that cover the dirty random writes. This reduces the total number of I/O requests but introduces writes for non-dirty data too. Therefore, DSCK introduces 8x and 4x additional writes for random writes and random read-write mix workloads leading to a runtime overhead of 9.5x and 2.6x respectively. For the random read-write mix workload,

28% additional reads are required by DSCK because our current implementation reads non-dirty file regions between dirty random writes.

**Cassandra:** We now measure the overhead imposed by DSCK on a Cassandra cluster from the client side. For this purpose, we run the YCSB benchmark with 4000 threads against a 3-node Cassandra cluster with the machine configuration as listed in Table 4.3. We run each Cassandra daemon as a single docker [128] container on each node. The 1.2 TB magnetic disk is used by the Cassandra daemon while the 480 GB SSD is used by DSCK for the corruption-resilient store. Because DSCK does not intercept library calls that perform reads, we use an insert-only workload that inserts 25 million entries. This workload is a worst-case scenario for DSCK as realistic workloads will contain reads as well. The average total runtime reported by the YCSB benchmark when running with and without DSCK are shown in Table 4.5. When running with DSCK, we use different configuration files that alter which files are protected using the corruption-resilient store. The average total runtime when using $\text{DSCK}_{\texttt{Cassandra}}$ with the default configuration is just 0.5% more than the average total runtime without DSCK. When using the default configuration, corrupted *Index* files are recovered in time proportional to their size by regenerating them from the corresponding *Data* file using the *scrub* tool. Their recovery times can be improved to under 5 minutes if the *Index* files are also protected in the corruption-resilient store; this configuration incurs a 25% runtime overhead under write-heavy workloads. Similarly, a configuration that also protects the *CommitLog* incurs a 45.4% runtime overhead but allows quick recovery of said files.

### 4.4.2 Corruption Resilience

We now use an experiment similar to the one we used earlier (Subsection 4.1.1) to evaluate the corruption resilience of the standard check and repair tool except for two differences: first, we run the DSCK check and repair tool for Cassandra instead of the

standard repair tool that comes with the distributed system and second, in every run, we corrupt multiple files belonging to the same file category instead of just one file to reduce the experiment time. The results, when using the default configuration detailed in Subsection 4.3.2, are shown in Table 4.6. The DSCK check and repair tool, when using the default configuration shown in Table 4.2, recovers all corrupted files except the *CommitLog* and improves corruption recovery to 89% of bytes stored on disk and to 99% of files stored on disk. When using a configuration that additionally protects the *Index* files similar corruption recovery results are achieved. If DSCK is run in a configuration where the *CommitLog* files are also protected (using the corruption-resilient store), then $\text{DSCK}_{\text{Cassandra}}$ can recover all the files and 100% of the bytes on disk. Alternately, if recovery time is not an issue, then the corrupted CommitLog files can be deleted and a full anti-entropy repair [57] can be run to recover any lost data.

### 4.4.3 Alternate Corruption Resilience Techniques

#### Replacing an Entire Node

When a node's disk state is corrupted and cannot be recovered using the repair tools that come with the distributed storage system, it can be replaced with a new node that is bootstrapped from other live replicas. Recovering lost data from other replicas takes significant time and reduces reliability. We measured the node replacement time for MongoDB, Cassandra, and Riak. The time taken to recover a single node is on the order of several hours for all systems (Table 4.7). In contrast, in many cases a small corruption can be repaired with a better check-and-recover tool. The table also shows the performance of $\text{DSCK}_{\text{Cassandra}}$ under four different configurations in repairing a local corruption within one node of the cluster. Instead of taking many hours to repair the node via full-store reconstruction from other nodes, $\text{DSCK}_{\text{Cassandra}}$ can repair many corruptions in minutes: most *Index* and *Data* files are repaired in under 30 minutes; all

| Setup | Files Recovered (%) | Data Recovered (%) |
|---|---|---|
| DSCK ("default" or "+index") | 192 (99%) | 544.5 MB (89%) |
| DSCK ("+log" or "+index,log") | 194 (100%) | 611.6 MB (100%) |

Table 4.6: **DSCK$_{Cassandra}$ corruption resilience.** *This table shows the number of files and the data on disk that DSCK$_{Cassandra}$ recovers after a corruption when used with the different configurations described in Subsection 4.3.2.*

| Setup | Description |
|---|---|
| Standard | MongoDB: 12.1 hrs or 2 hrs (using FS copy);<br>Cassandra: 6.1 hrs; Riak: 70 hrs |
| DSCK$_{Cassandra}$ (default) | <30 mins (most Index, Data corruptions);<br><5 mins (files in corruption-resilient store);<br>CommitLog files need full anti-entropy recovery. |
| DSCK$_{Cassandra}$ (+index) | <30 mins (most Data corruptions);<br><5 mins (Index, other files in corruption-resilient store);<br>CommitLog files need full anti-entropy recovery. |
| DSCK$_{Cassandra}$ (+log) | <30 mins (most Index, Data corruptions);<br><5 mins (CommitLog, other files in corruption-resilient store) |
| DSCK$_{Cassandra}$ (+index,log) | <30 mins (most Data corruptions);<br><5 mins (Index, CommitLog, other files in corruption-resilient store) |

Table 4.7: **Time to restore failed node.** *This table shows the time taken to replace a failed node using the other live replicas and the time taken by DSCK$_{Cassandra}$'s check and repair tool to fix the failed node.*

small system files protected using the corruption-resilient store are repaired in under 5 minutes.

#### Using File System's Corruption Resilience Features

File Systems like Btrfs and ZFS provide inbuilt RAID [149] protection. We now evaluate using Btrfs how its RAID protection features can be used to improve the corruption resilience of Cassandra. Modern NoSQL stores need to maintain replicas on separate machines spread across fault domains to handle various failure scenarios like power failures and hardware failures. Not keeping another local replica of the content that is already replicated remotely saves storage space and is preferable. Therefore, we craft a scheme that avoids locally replicating content that is already replicated on a remote machine. We partition two hard disk drives each into two 200 GB partitions. We then

create two Btrfs file systems each using a single partition from each of the two disks. In one Btrfs file system, which we call file system 1, we configure all data and metadata to be protected using RAID1 replication along with checksums. In the other Btrfs file system, which we call file system 2, we configure the data to not be replicated by choosing one of three different configurations: (a) we just use a single partition in which case there is a single copy for data but there are two copies for the metadata placed within a single partition; (b) we use two partitions and chose Btrfs's "single" option for data which spreads the files between the two partitions but uses RAID1 for metadata; or (c) we use two partitions and choose RAID0 striping across the two partitions for data but use RAID1 mirroring across the two partitions for metadata. It must be noted that in the file system 2, we ensure metadata always has two copies.

We store the on-disk folders maintained by Cassandra into one of the two Btrfs file systems and use symbolic links to create the file layout that Cassandra expects. We store the following folders in the Btrfs file system 2 that does not replicate data: "ycsb" folder that contains user stored content, "saved_caches" and "hints" folders that can be deleted if they get corrupted because they are needed for performance improvement and not correctness. All other folders are stored in the Btrfs file system 1 which protects everything including data using RAID1. Table 4.8 shows the time taken for inserting 25 million entries each of size approx. 1 KB using the YCSB benchmark application on the various Btrfs configurations. When using a single Btrfs file system without any corruption resilience, the time taken is 518.6 seconds. When using a configuration where the file system 2 just uses a single partition where only metadata is duplicated within the same partition, the time taken is 516.1 seconds. When the file system 2 is configured to store data in RAID0 striped fashion, the time taken for this workload is 391.5 seconds because of the increased disk parallelism due to striping. When the file system 2 is configured to store data in a "single" mode which allocates files between the two partitions, the time taken is 494.3 seconds.

| Btrfs Main FS 1 Configuration | Btrfs User Data FS 2 Configuration | Time Taken [Std. Dev.] (secs) |
|---|---|---|
| No replication at all | Not present | 518.6 [28] |
| All RAID1 | Single Partition Used | 516.1 [23] |
| All RAID1 | Data RAID0; Others RAID1 | 391.5 [4] |
| All RAID1 | Data single; Others RAID1 | 494.3 [31] |

| DSCK Results from Table 4.5 | Time Taken (secs) [Std. Dev.] |
|---|---|
| Without DSCK | 530.4 [17] |
| DSCK (default) | 533.1 [37] |
| DSCK (+index) | 662.5 [17] |
| DSCK (+log) | 771.3 [9] |
| DSCK (+index,log) | 774.2 [3] |

Table 4.8: **Corruption resilience through Btrfs.** *The top portion of the table shows the average time taken in seconds out of 8 trials along with the standard deviation for a YCSB workload benchmark that inserts 25 million entries each of size approx. 1 KB when using Btrfs file systems with varying replication configurations. For convenient comparison, the bottom portion of the table shows the time taken for the same workload with different configurations of DSCK copied over from Table 4.5.*

The Btrfs scrub tool allows manually checking for disk corruptions and repairing them using the replicas. When compared to DSCK, the Btrfs approach has both advantages and disadvantages. The advantage is that the performance of the Btrfs approach is very high because the replication and checksumming is integrated within the file system and is implemented very efficiently. The main disadvantage is that we need to create multiple file systems and we can only control at the folder level which files go into which file system. For example, this limitation is manifested in our configurations as not being able to replicate the very small auxiliary files associated with the user stored data. This is because these files are in the same folder as the files that store user stored data which are already replicated remotely and therefore we do not want them to be replicated locally as well. Another disadvantage with the Btrfs approach is that it takes away the freedom to use any file system.

## 4.5 Summary

We started this chapter by motivating the need for robust check and repair programs in distributed NoSQL stores. NoSQL stores are increasingly prevalent because they are linearly scalable. It is not uncommon to see thousand node deployments of such NoSQL stores. Such systems often maintain three or more replicas of the data in isolated fault domains to guarantee data availability in the presence of failures. However, the storage management toolchain is poorly developed in such NoSQL systems. For example, there is a lack of robust check and repair tools which are necessary in various scenarios like: during cluster-wide failures, to repair corrupted snapshots, and to quickly repair corruptions without requiring time-consuming full-node replacement from the other replicas. We also empirically show that the check and repair tools/techniques that come with three modern NoSQL stores – MongoDB, Cassandra and Riak – are very simplistic in design and do not handle data corruptions well.

We then take a "measure twice cut once" approach to this problem by first thoroughly studying the corruption resilient capability of the three NoSQL stores. We analyse these systems along the following questions: what semantic content do various files contain and how they are laid out hierarchically on the file system?, how hard is it to recover a corrupted file?, and what sort of data integrity protection is available to detect a corruption? We also specially look at certain *essential* files that are necessary for the system to start up. We make several observations from our study and state their implications for building a robust check and repair tool.

We then discuss the design of a framework, called DSCK, which enables implementing robust check and repair tools. DSCK consists of three main components - a corruption-resilient store, a checker and a repairer. The corruption-resilient store intercepts I/O related library calls made by the NoSQL system and classifies the I/O requests based on the destination file in order to transparently maintain checksummed

local replicas only for a selected set of files. Files that do not already have any data integrity protection from the NoSQL store or are hard to recover after a corruption are good candidates for protection through the corruption-resilient store. DSCK uses a configuration file with file-path prefixes, patterns and suffixes in order to guide the I/O classification. The checker relies on checksums – either those provided by the NoSQL store or the ones transparently added by DSCK – to detect corruptions using a file-type specific approach. The repairer uses several techniques to recover corrupted files. The appropriate recovery technique is chosen based on the type of file that is corrupted. Files protected through the corruption-resilient store are recovered from it. Data that is remotely replicated by the NoSQL store is recovered by fetching the corrupted content from the remote replica. Content that can be regenerated from other files using software are recovered using regeneration.

We then discuss the implementation and evaluation of $\text{DSCK}_{\text{Cassandra}}$, a check and repair tool for Cassandra, that imposes negligible overhead during normal operation but improves the corruption resilience tremendously. $\text{DSCK}_{\text{Cassandra}}$ is able to recover all files except the commit log files after corruption in the default configuration. A different configuration is able to recover the commit log files too but with approximately 45% overhead for a worst-case write only workload. Moreover, the recovery times for $\text{DSCK}_{\text{Cassandra}}$ is in the order of minutes while a full-node restore from replicas takes several hours.

# Chapter 5

# Related Work

Several researchers have worked on I/O classification in the past. The D-GRAID [188] RAID array classifies all I/O on semantically related data into a single class and places them in within a unit of fault containment. Using this strategy, D-GRAID achieves graceful degradation and quick-recovery in the presence of disk failures. X-RAY [22] uses graybox [12] techniques to classify data that is likely to be in the cache from the rest. Using this I/O classification, X-RAY implements an exclusive RAID cache that does not store data that is already present in the file system cache above it. Muthian et al. classify live blocks from dead blocks in a storage disk using two different techniques. They use this classification to implement a secure deleting disk that shreds logically deleted block so that deleted data can never be recovered. They also list out a range of other applications of classifying live blocks from dead blocks.

Self* [130] classifies files in an NFS server using machine learning techniques (decision trees). It exploits the strong associations between a file's properties and the names/attributes assigned to it. XN, the stable storage system for the Xok exokernel [101] also dealt with issues of classifying data blocks from metadata blocks. XN employed a template of metadata translation functions called *UDFs* specific to each file type. The responsibility of providing UDFs rested with the file system developer,

allowing the kernel to handle arbitrary metadata layouts without understanding the layout itself. We develop three new I/O classification techniques that are non-invasive and work under a variety of file systems without significant implementation effort.

## 5.1 David Related Work

Memulator [71] makes a great case for why storage emulation provides the unique ability to explore non-existent storage components and take end-to-end measurements. Memulator is a "timing-accurate" storage emulator that allows a simulated storage component to be plugged into a real system running real applications. Memulator can use the memory of either a networked machine or the local machine as the storage media of the emulated disk, enabling full system evaluation of hypothetical storage devices. Although this provides flexibility in device emulation, high-capacity devices requires an equivalent amount of memory; David provides the necessary scalability to emulate such devices. In turn, David can benefit from the networked-emulation capabilities of Memulator in scenarios when either the host machine has limited CPU and memory resources, or when the interference of running David on the same machine competing for the same resources is unacceptable.

One alternative to emulation is to simply buy a larger capacity or newer device and use it to run the benchmarks. This is sometimes feasible, but often not desirable. Even if one buys a larger disk, in the future they would need an even larger one; David allows one to keep up with this arms race without always investing in new devices. Note that we chose 1 TB as the upper limit for evaluation in this chapter because we could validate our results for that size. Having a large disk will also not address the issue of emulating much faster devices such as SSDs or RAID configurations. David emulates faster devices through an efficient use of memory as backing store.

Another alternative is to simulate the storage component under test; disk simulators

like Disksim [33] allow such an evaluation flexibly. However, simulation results are often far from perfect [63] – they fail to capture system dependencies and require the generation of representative I/O traces which is a challenge in itself. Finally, one might use analytical modeling for the storage devices; while very useful in some circumstances, it is not without its own set of challenges and limitations [184]. In particular, it is extremely hard to capture the interactions and complexities in real systems. Wherever possible, David does leverage well-tested analytical models for individual components to aid the emulation. Both simulation and analytical modeling are complementary to emulation, perfectly useful in their own right. Emulation does however provide a reasonable middle ground in terms of flexibility and realism.

Evaluation of how well an I/O system scales has been of interest in prior research and is becoming increasingly more relevant [228]. Chen and Patterson proposed a "self-scaling" benchmark that scales with the I/O system being evaluated, to stress the system in meaningful ways [40]. Although useful for disk and I/O systems, the self-scaling benchmarks are not directly applicable for file systems. The evaluation of the XFS file system from Silicon Graphics uses a number of benchmarks specifically intended to test its scalability [200]; such an evaluation can benefit from David to employ even larger benchmarks with greater ease; SpecSFS [220] also contains some techniques for scalable workload generation.

Similar to our emulation of scale in a storage system, Gupta et al. from UCSD propose a technique called *time dilation* for emulating network speeds orders of magnitude faster than available [74]. Time dilation allows one to experiment with unmodified applications running on commodity operating systems by subjecting them to much faster network speeds than actually available.

The classic text on disk drive modeling by Ruemmler and Wilkes [166] describes the different components of a disk drive in detail, and evaluates the ones that are necessary to model in order to achieve a high level of accuracy. While disk drive technology

and capacity have changed a lot since the paper was originally published, much of the underlying phenomena discussed then are still relevant.

Extraction of disk drive parameters has also been the subject of previous research to facilitate more accurate storage emulation. Skippy [203], developed by Talagala et al., is a tool for microbenchmark-based extraction of disk characteristics. Skippy linearly increases the stride while writing to the disk to factor out rotational effects from seek. Our disk model is optimized to run for large disks by introducing artificial delays between successive requests; a linear increase in stride is unacceptably slow for extracting parameters of large disks.

Worthington et al. describe techniques to extract disk drive parameters such as the seek profile, rotation time, and detailed information about disk layout and caching [222]. However, their techniques and the subsequent tool DIXtrac that automates the process, rely on the SCSI command interface [169], a limitation that is not acceptable since the majority of high capacity drives today use non-SCSI interfaces like IDE, ATA and SATA.

An orthogonal approach for disk modeling is to maintain runtime statistics in the form of a table, and use the information on past performance to predict the service times for future requests [11]. Popovici et al. develop the Disk Mimic [154], a table-based disk simulator that is embedded inside the I/O scheduler; in order to make informed scheduling decisions, the I/O scheduler performs on-line simulation of the underlying disk. One major drawback of table-based approaches is the amount of statistics that need to be maintained in order to deliver acceptable accuracy of prediction.

## 5.2 Sky Related Work

Mesnier et al. implement I/O classification by modifying the operating system and application to pass down classification information that can be used by the storage system for better caching [131]. IOFlow, a software defined storage architecture, classified

I/O requests at the VM granularity and enforced policies at various points in the I/O path [208]. Sonam et al. improve the performance of the *dmdedup* deduplication system by modifying the guest applications and file systems to generate hints [122]. In contrast, Sky obtains I/O-classification hints on a per system call basis without modifying the guest operating system or the file systems and reaps similar benefits. Sky also provides an equally expressive interface to modified I/O applications when compared to the above previous works as discussed in Subsection 3.2.6. However, Sky targets virtualized-storage in the context of VMM while the previous works are more broadly applicable.

Several caching algorithms have been proposed in the past such as LRU-K [143], ARC [127], 2Q [94], MQ [232], LRFU [114] etc. Our work on associating priorities with insights is complementary to these caching algorithms because these algorithms differentiate between disk blocks only based on their access patterns while Sky associates semantic meaning to the blocks. For example, Sky allows hits on data from a high-paying customer to be better than hits on data from a low-paying customer.

Several past research works have shown that insights can be gained by the storage systems using the knowledge of the on-disk layout of file systems [5, 186, 188, 189, 205]. Having more complex logic in the storage systems can make them less robust and more expensive. Sky generates insights with considerably lesser complexity in the storage.

Virtual Machine Introspection [79] in general and system-call interception specifically [53, 152] have been applied for malware analysis and other security applications in the past. LibVMI [118, 150] is a library to access the guest VM details that primarily supports memory accesses and events based on memory accesses. LibVMI has examples to show how to intercept system call entries but not system call exits. Sky uses system-call interception on both system call entry and exit for generating hints to improve storage performance. Sky introduces a new technique to intercept system call exits that use the IRET instruction in the Intel processors when compared to the past work on system-call interception using hardware extensions [53, 152]. Virtuoso [55] automatically generates

programs for accessing guest operating system information using training programs, trace collection and dynamic slicing. Techniques in Virtuoso could be used to fasten up certain parts of Sky like getting the PID of a process.

Roselli et al. use the auditing infrastructure in UNIX and the filter driver in Windows NT in order to collect and analyze traces to understand different file system workloads (e.g. block lifetimes) [161]. Sky obtains similar information about the block lifetimes through system-call interception. FADED [186] provides secure file deletes by providing block liveness to the storage device. It detects file deletes and truncates implicitly by tracking file system on-disk data structures and also making small modification to file systems when necessary. Sky directly intercepts `unlink`, `truncate` and related system calls to know about file deletes and truncates. Because Sky uses checksums, there is a loss of accuracy in rare scenarios when file system metadata and data content generate the same checksums. A more sophisticated future version of Sky could avoid this inaccuracy by using file system knowledge.

VirtFS is a paravirtualized file system that avoids the overheads associated with a generic networked file system by leveraging the 9p distributed file system protocol directly on top of a paravirtualized transport [100]. Sky could be used with VirtFS in order to allow guest applications to pass hints to VMM without modifying the 9p protocol and the VirtIO transport.

Gu et al. bridge the semantic gap between a VM and the VMM by running a process from the host on the guest VM under the cover of an existing running process in the guest [72]. Such an approach will be costly for Sky because intercepting system calls at userspace level is expensive due to the kernel boundary crossings and the context switches between the monitored and monitoring processes.

Geiger [96] explores techniques to passively infer useful information about guest operating system's unified buffer cache and virtual memory system in order to provide eviction based cache placement. Sky is complementary to Geiger and gives higher prior-

ity to small files, file system metadata and application classified higher priority content. Antfarm [95] describes techniques used by the VMM for tracking the existence and activities of guest operating system processes without detailed knowledge of a guest's internal architecture or implementation. Sky tracks threads in addition to processes. Moreover, Sky intercepts system calls to track I/O performed by processes.

## 5.3 DSCK Related Work

Many past research works have studied the corruption resilience of storage systems. Ganesan et al. studied the effects of corruption on eight modern distributed storage systems and find that they do not consistently use redundancy to recover from file-system errors [62]. Another observation from their work is that modern distributed systems often crash when handling corruption locally leading to more severe global cluster-wide effects. Lakshmi et al. use type-aware pointer corruption to evaluate the corruption resilience of Windows NTFS and Linux Ext3 file systems [21]. They find that both these file systems do not recover from most corruptions including those scenarios where there is enough redundant information to perform recovery. Sriram et al. inject faults into the MySQL [145] DBMS system and find that corruptions can lead to untimely crashes, data loss and incorrect results [195]. All these systems show poor corruption resilience. We also find poor corruption resilience in the check and repair tools that ship with the modern distributed storage systems that we investigated in this paper.

Marinescu et al. intercept library calls made by an application using the LD_PRELOAD technique to inject faults [123]. They build a tool called LFI that automatically prepares fault scenarios and use them to inject faults. Patterson et al. build a tool called FIG (Fault injection in glibc) for injecting faults at the application/system boundary. DSCK uses the same interception technique as previous works but for a different purpose: to maintain corruption-resilient replicas for a selected set of files. Maintaining replicas

of files for performance and reliability is a widely used technique in several file and storage systems [31, 149, 159]. DSCK maintains local replicas of a selected set of files along with checksums and protects them from the effects of corruption.

File systems have had check and repair tools for a long time [29, 37, 120, 125, 126]. Such tools often use the redundancy of certain file system structures to recover from a corruption. DSCK also maintains local replicas of the distributed storage system's metadata in order to enhance corruption-resilience and recovery.

Many systems have selectively replicated metadata to enhance performance and reliability. D-GRAID [188] is a RAID storage array that selectively replicates semantically critical data to ensure that the storage array gracefully degrades in the presence of failures. IRON file system [156] also selectively replicates metadata in order to improve the robustness in the presence of failures. Gnothi [218] is a block-replication system that selectively replicates metadata to enhance availability and failure recovery. PARTE [117] is a parallel file system that replicates and distributes a file's metadata in order to provide high availability. DSCK also selectively replicas the distributed storage system's local node metadata to improve availability in the presence of failures.

# Chapter 6

# Conclusion and Future Work

This dissertation focused on "non-invasive I/O classification" as a means to improve storage systems by developing novel easily deployable applications. We developed three new non-invasive I/O classification techniques that further work with many different file systems without the need for significant additional effort. Our contributions with the three novel applications are in diverse but important areas like benchmarking, virtualized storage and corruption-resilience of distributed NoSQL storage. We believe that our applications are more widely adoptable when compared to other solutions that require extensive modifications to one or more components in the storage stack to make them work with a particular storage configuration.

The first novel application we develop is a novel benchmarking emulator called David that removes the frustration in doing large-scale experimentation on realistic storage hardware – a problem many in the storage community face. David makes it practical to experiment with benchmarks that were otherwise infeasible to run on a given system, by transparently scaling down the storage capacity required to run the workload. David creates a "compressed" version of the original file-system image by omitting all file data and laying out metadata more efficiently; an online storage model determines the runtime of the benchmark workload on the original uncompressed image.

We showed that David reduces storage requirements by orders of magnitude; David is able to emulate a 1 TB target workload using only an 80 GB available disk, and predicts the actual runtime accurately. David can also emulate newer or faster devices, e.g., we showed how David can effectively emulate a multi-disk RAID and a futuristic SSD disk using a limited amount of memory. David works under any file system as demonstrated in this paper with Ext3 and Btrfs.

Our second contribution is a smart caching and deduplication system in the hypervisor that classifies I/O requests and treats them differentially – file system metadata and small files get higher priority in the cache, encrypted file content and file-copy are handled efficiently during deduplication. We achieve this using Sky, an extension to the VMM that gathers insights and information by intercepting system calls made by guest applications. We use system-call interception as a core technique so that a VMM can gather insights and information without requiring modifications to the guest operating system or the guest application. Sky uses system-call interception to gather insights about guest application issues I/O requests. We showed through experiments that system-call interception is an efficient way to obtain useful insights about I/O-bound guest applications with minimal overheads (under 5%). We showed how Sky gains three specific insights – guest file-size information, metadata-data distinction, and file-content hints – and uses said information to enhance virtualized-storage performance. By caching small files and metadata with higher priority, Sky reduces the runtime by 2.3 to 8.8 times for certain workloads. Sky also achieves 4.5 to 18.7 times reduction in the runtime of an open-source block-layer deduplication system by exploiting hints about file contents. Sky works underneath both Linux and FreeBSD guests, as well as under a range of file systems, thus enabling portable and general VMM-level optimization underneath a wide range of storage stacks.

Our third contribution enhances the corruption-resilience of the check and repair tools that come with modern distributed NoSQL stores. Check-and-recover tools have

long formed a core part of the storage management toolchain. As we have shown, modern NoSQL systems do not yet have such robust check-and-recover, and thus a gap exists in how such systems are managed. We introduced DSCK, a generic framework to aid in the construction of check-and-recover tools for such systems. Our analysis of the check and repair tools that come with MongoDB, Cassandra, and Riak showed that they are not resilient to various data corruptions. The results of our analysis directly feed into the design of DSCK, which provides a local corruption-resilient store, an I/O interception library, and a base check/repair framework, all of which can be customized to meet the needs of a specific storage system. We have implemented $\text{DSCK}_{Cassandra}$, a full-fledged check-and-recover tool for Cassandra. $\text{DSCK}_{Cassandra}$ improves corruption recoverability from 37.5% of the files to nearly 100% of the files stored by Cassandra while inducing little to modest performance overheads. We also showed that using $\text{DSCK}_{Cassandra}$ to repair certain file corruptions enables full-node restore in minutes rather than hours.

## 6.1 Learnings

We now describe some of our learnings from the systems we build towards this dissertation.

**Non-Invasive techniques are possible:** With careful thought, it is feasible to develop non-invasive, easy to adopt techniques that can provide the same benefits as their invasive counterparts. We showed three examples in this dissertation where a non-invasive design was chosen in preference over an invasive approach. Depending on the final application being built, one has to be careful about the overheads imposed by the non-invasive approach.

**I/O classification is easier higher up in the storage stack:** There is more information higher up in the storage stack making it is easier to classify I/O requests. More importantly, classifying I/O requests higher up can often tolerate changes in the storage components below it in the storage hierarchy. For example, implementing DSCK by intercepting library calls and implementing David and Sky by implementing system calls makes our techniques tolerate different file systems.

**Innovators can explore farther when unconstrained by physical resources:** Procurement of physical resources can hamper innovations; especially if the resources are in the design stage. We were able to explore a new disk interface called "nameless writes" that helps lower the cost of huge capacity SSD drives with the aid of David emulator tool. We would not have been able to perform this innovation if we did not have David.

**Hardware mechanisms are well suited for novel applications:** Hardware mechanisms are a very powerful means to implement novel applications. They impose low overheads and are well tested for robustness. We used the hardware mechanisms in the modern processors with virtualization extensions to intercept system call entry and exit in order to gather insights with Sky.

**The uncommon failure scenarios don't get the attention they deserve:** Uncommon failures in distributed storage can cause catastrophic damages like permanent data loss or several days of service interruption. The financial impact of such catastrophic failures can be huge [8, 24, 44, 46, 50, 51, 59, 68, 105, 207, 227]. The check and repair tools in the distributed storage systems studied in Chapter 4 do not safely handle the uncommon scenario of corruption to critical system files. They lose access to existing user data or need the help of an expert to salvage the uncorrupted data. In rare scenarios, such errors can lead to much more serious failures like multi-hour service outages. The repair tools in LevelDB and RocksDB lower level database engines handle corruption by archiving

the affected SST files in their entirety. Such archived SST files often have uncorrupted data too. Instead, these tools need to try and recover the lost data along with cooperation from the higher level distributed NoSQL stores.

## 6.2 Future Work

In this section, we list out opportunities for future research work related to this dissertation.

**Speed-up mode for David:** It will be beneficial for the file system benchmarking community to make David work in a *speed-up* mode that resembles simulation; either completely or partially during certain phases of the workload. The *speed-up* mode can significantly reduce the total benchmark time. However, the challenge is to accurately predict the benchmark runtime.

**Develop Storage Models for more devices:** We have emulated a magnetic disk, a simple software RAID-1 device and an SSD device using David. In the future, it will be a worthwhile exercise to make David emulate more devices including complex hardware RAID configurations. This will make David a versatile and widely usable tool.

**Sky for windows operating systems:** Sky currently supports the Linux and FreeBSD operating systems. Making Sky support a much different operating system like Windows will make it more universal. We expect some new challenges here because Windows differs significantly from the Unix-flavored operating systems.

**Evaluate workloads that use `mmap` exhaustively with Sky:** Although Sky handles memory-mapped workloads, our evaluation did not extensively test this. In the future,

evaluating applications like database servers that exhaustively use `mmap` will add additional data points to the evaluation of Sky.

**Alternate techniques for use with Sky:** There is scope for evaluating three alternate techniques for use with Sky: (1) Sky currently uses insight-calls to read the guest operating system state. In the future, Sky could explore other techniques [55] that can be used to fetch the required information directly from the guest operating system's memory. (2) Sky uses executable names and file extensions to detect applications that encrypt files. In the future, more sophisticated techniques that examine the contents of the executable binary as well as the file contents could be developed. (3) Our current prototype Sky implementation uses a configuration parameter to identify the guest operating system. A future version of Sky could identify the guest operating system by using VM memory analysis [118] and details about system calls like the numbers, arguments, return values and frequencies.

**Further decrease the system-call interception overheads:** The overheads due to system-call interception in Sky are low enough for real I/O workloads (under 5%). This overhead could be reduced with further research so that Sky is useful for a wider variety of applications.

**Gather more insights using Sky:** Exploring more insights for improving the virtualized-storage or for achieving broader goals is a promising research direction. Sky, being part of the hypervisor, is in a unique powerful position to gather insights with less difficulty.

**Corruption resilient check and repair for MongoDB and Riak:** We used DSCK to implement a corruption resilient check and repair tool for Cassandra called $\text{DSCK}_{\text{Cassandra}}$. Implementing similar tools for MongoDB, Riak and other distributed NoSQL stores will definitely prove useful in improving their corruption resilience.

## 6.3 Summary

I/O classification is a well known technique that has a wide range of applications [22, 117, 131, 188, 218]. This dissertation emphasizes the need for deployable, non-invasive I/O classification techniques that work in different configurations without significant additional effort. Our approach avoids the need to change existing interfaces or components in a storage stack unless it is absolutely necessary. We believe that our techniques lead to easier adoption in existing storage environments. We built three diverse, novel applications using such I/O classification techniques.

David, our first application, allows a storage researcher to accurately benchmark futuristic, huge capacity storage disks using much smaller available storage. David classifies I/O requests containing file system metadata from those that contain application data in order to reduce the amount of physical storage needed. Sky, our second contribution, gathers insights about guest I/O applications in a virtual machine using system-call interception. We use these insights to classify I/O requests and treat them differentially at the hypervisor level to implement smart caching and deduplication systems. Finally, we built DSCK, a framework for implementing corruption-resilient check and repair tools in modern distributed NoSQL stores. $\text{DSCK}_{\text{Cassandra}}$, our corruption-resilient check and repair tool for Cassandra, imposes modest overheads but improves corruption-resilience and recovery times.

# Bibliography

[1] Abutalib Aghayev and Peter Desnoyers. Skylight—A Window on Shingled Disk Operation. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 135–149, Santa Clara, CA, 2015. USENIX Association.

[2] Abutalib Aghayev, Theodore Ts'o, Garth Gibson, and Peter Desnoyers. Evolving Ext4 for Shingled Disks. In *Proceedings of the 15th Usenix Conference on File and Storage Technologies*, FAST'17, pages 105–119, Berkeley, CA, USA, 2017. USENIX Association.

[3] Nitin Agrawal. *Representative, Reproducible, and Practical Benchmarking of File and Storage Systems*. PhD thesis, University of Wisconsin-Madison, August 2009.

[4] Nitin Agrawal, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Generating Realistic Impressions for File-System Benchmarking.

[5] Nitin Agrawal, Leo Arulraj, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Emulating Goliath Storage Systems with David. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies*, FAST'11, Berkeley, CA, USA, 2011. USENIX Association.

[6] Nitin Agrawal, William J. Bolosky, John R. Douceur, and Jacob R. Lorch. A Five-Year Study of File-System Metadata.

[7] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D. Davis, Mark Manasse, and Rina Panigrahy. Design Tradeoffs for SSD Performance. In *Proceedings of the Usenix Annual Technical Conference (USENIX '08)*, Boston, MA, June 2008.

[8] Ramnatthan Alagappan, Aishwarya Ganesan, Yuvraj Patel, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Correlated Crash Vulnerabilities. In

*Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 151–167, Berkeley, CA, USA, 2016. USENIX Association.

[9] AMD Technology. AMD64 Architecture Programmer's Manual Volume 2: System Programming. `http://developer.amd.com/wordpress/media/2012/10/24593_APM_v21.pdf`, 2012.

[10] Dave Anderson, Jim Dykes, and Erik Riedel. More Than an Interface—SCSI vs. ATA. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, FAST '03, pages 245–257, Berkeley, CA, USA, 2003. USENIX Association.

[11] Eric Anderson. Simple table-based modeling of storage devices. Technical Report HPL-SSP-2001-04, HP Laboratories, July 2001.

[12] Andrea C. Arpaci-Dusseau and Remzi H. Arpaci-Dusseau. Information and Control in Gray-box Systems. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01, pages 43–56, New York, NY, USA, 2001. ACM.

[13] Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Lakshmi N. Bairavasundaram, Timothy E. Denehy, Florentina I. Popovici, Vijayan Prabhakaran, and Muthian Sivathanu. Semantically-smart Disk Systems: Past, Present, and Future. *SIGMETRICS Perform. Eval. Rev.*, 33(4):29–35, March 2006.

[14] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau. Fail-Stutter fault tolerance. In *Proceedings Eighth Workshop on Hot Topics in Operating Systems*, pages 33–38, May 2001.

[15] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 0.91 edition, May 2015.

[16] Leo Arulraj, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Improving Virtualized Storage Performance with Sky. In *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '17, pages 112–128, New York, NY, USA, 2017. ACM.

[17] Jens Axboe. Flexible I/O Tester. `https://github.com/axboe/fio`, 2018.

[18] Eitan Bachmat and Jiri Schindler. Analysis of Methods for Scheduling Low Priority Disk Drive Tasks. In *Proceedings of the 2002 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '02, pages 55–65, New York, NY, USA, 2002. ACM.

[19] Lakshmi N. Bairavasundaram, Garth R. Goodson, Shankar Pasupathy, and Jiri Schindler. An Analysis of Latent Sector Errors in Disk Drives. In *Proceedings of the International Conference on Measurements and Modeling of Computer Systems (SIGMETRICS'07)*, San Diego, California, June 2007.

[20] Lakshmi N. Bairavasundaram, Garth R. Goodson, Bianca Schroeder, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. An Analysis of Data Corruption in the Storage Stack. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST '08)*, San Jose, California, February 2008.

[21] Lakshmi N. Bairavasundaram, Meenali Rungta, Nitin Agrawal, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Michael M. Swift. Analyzing the Effects of Disk-Pointer Corruption. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN'08)*, Anchorage, Alaska, June 2008.

[22] Lakshmi N. Bairavasundaram, Muthian Sivathanu, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. X-RAY: A Non-Invasive Exclusive Caching Mechanism for RAIDs. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, ISCA '04, Washington, DC, USA, 2004. IEEE Computer Society.

[23] Mary Baker, Mehul Shah, David S. H. Rosenthal, Mema Roussopoulos, Petros Maniatis, TJ Giuli, and Prashanth Bungale. A Fresh Look at the Reliability of Long-term Digital Storage. *SIGOPS Oper. Syst. Rev.*, 40(4):221–234, April 2006.

[24] Mehmet Bakkaloglu, Jay J Wylie, Chenxi Wang, and Gregory R Ganger. On Correlated Failures in Survivable Storage Systems . Technical report, DTIC Document, 2002.

[25] P. R. Barham. A fresh approach to file system quality of service. In *Network and Operating System Support for Digital Audio and Video, 1997., Proceedings of the IEEE 7th International Workshop on*, pages 113–122, May 1997.

[26] W. Bartlett and L. Spainhower. Commercial fault tolerance: a tale of two systems. *IEEE Transactions on Dependable and Secure Computing*, 1(1):87–96, Jan 2004.

[27] Fabrice Bellard. QEMU, a Fast and Portable Dynamic Translator. In *USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.

[28] Alberto Bertogli. libjio - A library for Journaled I/O. `https://blitiri.com.ar/p/libjio/`.

[29] E.J. Bina and P.A. Emrath. *A Faster Fsck for BSD UNIX*. Report. University of Illinois at Urbana-Champaign, Center for Supercomputing Research and Development, 1988.

[30] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The Gem5 Simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, August 2011.

[31] Jeff Bonwick, Matt Ahrens, Val Henson, Mark Maybee, and Mark Shellenbaum. The zettabyte file system. In *Proc. of the 2nd Usenix Conference on File and Storage Technologies*, volume 215, 2003.

[32] T. C. Bressoud and F. B. Schneider. Hypervisor-based Fault Tolerance. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 1–11, New York, NY, USA, 1995. ACM.

[33] John S. Bucy, Jiri Schindler, Steven W. Schlosser, and Gregory R. Ganger. The DiskSim Simulation Environment Version 4.0 Reference Manual. Technical Report CMU-PDL-08-101, Carnegie Mellon University, May 2008.

[34] Edouard Bugnion, Scott Devine, and Mendel Rosenblum. Disco: Running Commodity Operating Systems on Scalable Multiprocessors. pages 143–156.

[35] W. Burleson, O. Mutlu, and M. Tiwari. Invited: Who is the major threat to tomorrow's security? You, the hardware designer. In *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–5, June 2016.

[36] Josiah L. Carlson. *Redis in Action*. Manning Publications Co., Greenwich, CT, USA, 2013.

[37] João Carlos Menezes Carreira, Rodrigo Rodrigues, George Candea, and Rupak Majumdar. Scalable Testing of File System Checkers. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 239–252, New York, NY, USA, 2012. ACM.

[38] Adrian M. Caulfield, Laura M. Grupp, and Steven Swanson. Gordon: Using Flash Memory to Build Fast, Power-efficient Clusters for Data-intensive Applications. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, pages 217–228, New York, NY, USA, 2009. ACM.

[39] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A Distributed Storage System for Structured Data. *ACM Trans. Comput. Syst.*, 26(2):4:1–4:26, June 2008.

[40] Peter M. Chen and David A. Patterson. A New Approach to I/O Performance Evaluation–Self-Scaling I/O Benchmarks, Predicted I/O Performance. pages 1–12.

[41] P.M. Chen and B.D. Noble. When virtual is better than real [operating system relocation to virtual machines]. In *Hot Topics in Operating Systems, 2001. Proceedings of the Eighth Workshop on*, pages 133–138, 2001.

[42] Kristina Chodorow. *MongoDB: The Definitive Guide: Powerful and Scalable Data Storage*. O'Reilly, 2nd edition, 2013.

[43] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An Empirical Study of Operating Systems Errors. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01, pages 73–88, New York, NY, USA, 2001. ACM.

[44] Asaf Cidon, Stephen M. Rumble, Ryan Stutsman, Sachin Katti, John Ousterhout, and Mendel Rosenblum. Copysets: Reducing the Frequency of Data Loss in Cloud Storage. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, USENIX ATC'13, pages 37–48, Berkeley, CA, USA, 2013. USENIX Association.

[45] Clutch. Best Cloud Service Providers. `https://clutch.co/cloud`, 2017.

[46] ComputerWorldUK. Lightning strikes Amazon and Microsoft data centres. `http://www.computerworlduk.com/galleries/infrastructure/ten-datacentre-disasters-that-brought-firms-offline-3593580/#5`.

[47] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM.

[48] Peter Corbett, Bob English, Atul Goel, Tomislav Grcanac, Steven Kleiman, James Leong, and Sunitha Sankar. Row-diagonal Parity for Double Disk Failure Correction. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, FAST'04, pages 1–14, Berkeley, CA, USA, 2004. USENIX Association.

[49] Brendan Cully, Geoffrey Lefebvre, Dutch Meyer, Mike Feeley, Norm Hutchinson, and Andrew Warfield. Remus: High Availability via Asynchronous Virtual Machine Replication. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'08, pages 161–174, Berkeley, CA, USA, 2008. USENIX Association.

[50] DataCenterDynamics. Lessons from the Singapore Exchange failure. `http://www.datacenterdynamics.com/power-cooling/lessons-from-the-singapore-exchange-failure/94438.fullarticle`.

[51] DataCenterKnowledge. Lightning Disrupts Google Cloud Services. `http://www.datacenterknowledge.com/archives/2015/08/19/lightning-strikes-google-data-center-disrupts-cloud-services/`.

[52] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-value Store. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 205–220, New York, NY, USA, 2007. ACM.

[53] Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee. Ether: malware analysis via hardware virtualization extensions. In *Proceedings of the 15th ACM conference on Computer and communications security*, CCS '08, pages 51–62, New York, NY, USA, 2008. ACM.

[54] Xiaoning Ding, Song Jiang, Feng Chen, Kei Davis, and Xiaodong Zhang. DiskSeen: Exploiting Disk Layout and Access History to Enhance I/O Prefetch. In *Proc. of USENIX'07*, 2007.

[55] B. Dolan-Gavitt, T. Leek, M. Zhivich, J. Giffin, and Wenke Lee. Virtuoso: Narrowing the Semantic Gap in Virtual Machine Introspection. In *Security and Privacy (SP), 2011 IEEE Symposium on*, pages 297–312, 2011.

[56] J. G. Elerath and S. Shah. Server class disk drives: how reliable are they? In *Annual Symposium Reliability and Maintainability, 2004 - RAMS*, pages 151–156, Jan 2004.

[57] Datastax Enterprise. Manual repair: Anti-entropy repair. `http://docs.datastax.com/en/cassandra/3.0/cassandra/operations/opsRepairNodesManualRepair.html`, 2017.

[58] Datastax Enterprise. Read Repair: repair during read path. `http://docs.datastax.com/en/cassandra/3.0/cassandra/operations/opsRepairNodesReadRepair.html`, 2017.

[59] Daniel Ford, François Labelle, Florentina I. Popovici, Murray Stokely, Van-Anh Truong, Luiz Barroso, Carrie Grimes, and Sean Quinlan. Availability in Globally Distributed Storage Systems. In *Presented as part of the 9th USENIX Symposium on Operating Systems Design and Implementation*, Berkeley, CA, 2010. USENIX.

[60] Glenn Fowler, Landon C. Noll, and Phong Vo. Fowler / Noll / Vo (FNV) Hash, 1991.

[61] Daniel Fryer, Mike Qin, Jack Sun, Kah Wai Lee, Angela Demke Brown, and Ashvin Goel. Checking the Integrity of Transactional Mechanisms. *Trans. Storage*, 10(4):17:1–17:23, October 2014.

[62] Aishwarya Ganesan, Ramnatthan Alagappan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Redundancy Does Not Imply Fault Tolerance: Analysis of Distributed Storage Reactions to Single Errors and Corruptions. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 149–166, Santa Clara, CA, 2017. USENIX Association.

[63] Gregory R. Ganger and Yale N. Patt. Using system-level models to evaluate i/o subsystem designs. *IEEE Trans. Comput.*, 47(6):667–678, 1998.

[64] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: A Virtual Machine-based Platform for Trusted Computing. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 193–206, New York, NY, USA, 2003. ACM.

[65] Tal Garfinkel, Mendel Rosenblum, et al. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *NDSS*, volume 3, pages 191–206, 2003.

[66] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 29–43, New York, NY, USA, 2003. ACM.

[67] Garth A. Gibson, David Rochberg, Jim Zelenka, David F. Nagle, Khalil Amiri, Fay W. Chang, Eugene M. Feinberg, Howard Gobioff, Chen Lee, Berend Ozceri, and Erik Riedel. File server scaling with network-attached secure disks. pages 272–284.

[68] Google. Google Cloud Status. `https://status.cloud.google.com/incident/compute/15056#5719570367119360`.

[69] Jim Gray. What Next?: A Dozen Information-technology Research Goals. *J. ACM*, 50(1):41–57, January 2003.

[70] Jim Gray and Catharine van Ingen. Empirical Measurements of Disk Failure Rates and Error Rates. *CoRR*, abs/cs/0701166, 2007.

[71] John Linwood Griffin, Jiri Schindler, Steven W. Schlosser, John S. Bucy, and Gregory R. Ganger. Timing-accurate Storage Emulation.

[72] Zhongshu Gu, Zhui Deng, Dongyan Xu, and Xuxian Jiang. Process Implanting: A New Active Introspection Framework for Virtualization. In *Proceedings of the 2011 IEEE 30th International Symposium on Reliable Distributed Systems*, SRDS '11, pages 147–156, Washington, DC, USA, 2011. IEEE Computer Society.

[73] Haryadi S. Gunawi, Vijayan Prabhakaran, Swetha Krishnan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Improving File System Reliability with I/O Shepherding. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 293–306, New York, NY, USA, 2007. ACM.

[74] Diwaker Gupta, Kenneth Yocum, Marvin McNett, Alex C. Snoeren, Amin Vahdat, and Geoffrey M. Voelker. To infinity and beyond: time-warped network emulation. In *Proceedings of the 3rd conference on Networked Systems Design and Implementation ( NSDI'06 )*, San Jose, CA, 2006.

[75] S. Gurumurthi, A. Sivasubramaniam, M. Kandemir, and H. Franke. DRPM: dynamic speed control for power management in server class disks. In *30th Annual International Symposium on Computer Architecture, 2003. Proceedings.*, pages 169–179, June 2003.

[76] Andreas Haeberlen, Alan Mislove, and Peter Druschel. Glacier: Highly Durable, Decentralized Storage Despite Massive Correlated Failures. In *Proceedings of the 2Nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2*, NSDI'05, pages 143–158, Berkeley, CA, USA, 2005. USENIX Association.

[77] J. L. Hafner, V. Deenadhayalan, W. Belluomini, and K. Rao. Undetected disk errors in RAID arrays. *IBM Journal of Research and Development*, 52(4.5):413–425, July 2008.

[78] Tom's Hardware. The Mother of All CPU Charts 2005/2006. `http://www.tomshardware.com/reviews/mother-cpu-charts-2005,1175.html`, 2005.

[79] Y. Hebbal, S. Laniepce, and J. M. Menaud. Virtual Machine Introspection: Techniques and Applications. In *2015 10th International Conference on Availability, Reliability and Security*, pages 676–685, Aug 2015.

[80] Dave Hitz, James Lau, and Michael A Malcolm. File System Design for an NFS File Server Appliance. In *USENIX winter*, volume 94, 1994.

[81] Hai Huang, A Hung, and Kang G. Shin. FS2: dynamic data replication in free disk space for improving disk performance and energy consumption. In *Proceedings of 20th ACM Symposium on Operating System Principles*, pages 263–276. ACM Press, 2005.

[82] Basho Technologies Inc. Backing Up. `https://www.tiot.jp/riak-docs/riak/kv/2.0.2/using/cluster-operations/backing-up/`, 2017.

[83] Basho Technologies Inc. Riak Customers. `http://basho.com/about/customers/`, 2017.

[84] Basho Technologies Inc. Riak KV Repair Techniques. `https://github.com/basho/basho_docs/blob/master/content/riak/kv/2.2.3/using/repair-recovery/repairs.md`, 2017.

[85] DataStax Inc. About snapshots. `https://docs.datastax.com/en/cassandra/3.0/cassandra/operations/opsAboutSnapshots.html?hl=backup`, 2017.

[86] DataStax Inc. Cassandra Customers. `https://www.datastax.com/customers`, 2017.

[87] DataStax Inc. Cassandra Repair Tools: 'sstablescrub' and 'nodetool scrub'. `http://docs.datastax.com/en/cassandra/3.0/cassandra/tools/toolsSSTableScrub.html?hl=sstablescrub` `http://docs.datastax.com/en/cassandra/3.0/cassandra/tools/toolsScrub.html?hl=nodetool%2Cscrub`, 2017.

[88] MongoDB Inc. MongoDB Backup Methods. `https://docs.mongodb.com/manual/core/backups/`, 2017.

[89] MongoDB Inc. MongoDB Customers. `https://www.mongodb.com/who-uses-mongodb`, 2017.

[90] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual Combined Volumes: 1, 2ABC, 3ABCD. https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf, 2016.

[91] Laurence James. 10 years of Kryder's Law: how flash will shape the next 10 years. https://thestack.com/data-centre/2015/12/09/10-years-of-kryders-law-how-flash-will-shape-the-next-10-years/, 2015.

[92] Weihang Jiang, Chongfeng Hu, Yuanyuan Zhou, and Arkady Kanevsky. Are Disks the Dominant Contributor for Storage Failures?: A Comprehensive Study of Storage Subsystem Failure Characteristics. *Trans. Storage*, 4(3):7:1–7:25, November 2008.

[93] Heeseung Jo, Youngjin Kwon, Hwanju Kim, Euiseong Seo, Joonwon Lee, and Seungryoul Maeng. SSD-HDD-hybrid Virtual Disk in Consolidated Environments. In *Proceedings of the 2009 International Conference on Parallel Processing*, Euro-Par'09, pages 375–384, Berlin, Heidelberg, 2010. Springer-Verlag.

[94] Theodore Johnson and Dennis Shasha. 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, pages 439–450, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[95] Stephen T. Jones, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Antfarm: Tracking Processes in a Virtual Machine Environment. In *in Proc. of the USENIX Annual Technical Conf*, 2006.

[96] Stephen T. Jones, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Geiger: monitoring the buffer cache in a virtual machine environment. *SIGARCH Comput. Archit. News*, 34(5):14–24, October 2006.

[97] Stephen T. Jones, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. VMM-based Hidden Process Detection and Identification Using Lycosid. In *Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '08, pages 91–100, New York, NY, USA, 2008. ACM.

[98] William K. Josephson, Lars A. Bongo, Kai Li, and David Flynn. DFS: A File System for Virtualized Flash Storage. *Trans. Storage*, 6(3):14:1–14:25, September 2010.

[99] Ashok Joshi, Sam Haradhvala, and Charles Lamb. Oracle NoSQL databasescalable, transactional key-value store. In *Proc. the 2nd International Conference on Advances in Information Mining and Management*, pages 75–78, 2012.

[100] V. Jujjuri, E. Van Hensbergen, A. Liguori, and B. Pulavarty. VirtFS – A virtualization aware File System pass-through. In *Proceedings of the Ottawa Linux Symposium*, 2010.

[101] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Héctor Briceño, Russell Hunt, David Mazières, Thomas Pinckney, Robert Grimm, John Jannotti, and Kenneth Mackenzie. Application Performance and Flexibility on Exokernel Systems. pages 52–65.

[102] H. H. Kari, H. Saikkonen, and F. Lombardi. Detection of defective media in disks. In *Proceedings of 1993 IEEE International Workshop on Defect and Fault Tolerance in VLSI Systems*, pages 49–55, Oct 1993.

[103] Jeffrey Katcher. Postmark Source Code. `http://linuxcompressed.sourceforge.net/linux24-cc/statistics/testsuite/postmark-1_5.c`, 2007.

[104] Atish Kathpal and Priya Sehgal. BARNS: Towards Building Backup and Recovery for NoSQL Databases. In *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 17)*, Santa Clara, CA, 2017. USENIX Association.

[105] Kimberley Keeton, Cipriano Santos, Dirk Beyer, Jeffrey Chase, and John Wilkes. Designing for Disasters. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, FAST '04, pages 59–62, Berkeley, CA, USA, 2004. USENIX Association.

[106] Kimberly Keeton and John Wilkes. Automating Data Dependability. In *Proceedings of the 10th Workshop on ACM SIGOPS European Workshop*, EW 10, pages 93–100, New York, NY, USA, 2002. ACM.

[107] Robert W. Kembel. *Fibre Channel A Comprehensive Introduction*. Northwest Learning Assoc, 2009.

[108] The kernel development community. Submitting patches: the essential guide to getting your code into the kernel. `https://www.kernel.org/doc/html/v4.12/process/submitting-patches.html`, 2017.

[109] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. KVM: the Linux virtual machine monitor. In *Proceedings of the Linux Symposium*, volume 1, pages 225–230, 2007.

[110] Rusty Klophaus. Riak Core: Building Distributed Applications Without Shared State. In *ACM SIGPLAN Commercial Users of Functional Programming*, CUFP '10, New York, NY, USA, 2010. ACM.

[111] Ryusuke Konishi, Yoshiji Amagai, Koji Sato, Hisashi Hifumi, Seiji Kihara, and Satoshi Moriai. The Linux Implementation of a Log-structured File System. *SIGOPS Oper. Syst. Rev.*, 40(3):102–107, July 2006.

[112] Harendra Kumar, Yuvraj Patel, Ram Kesavan, and Sumith Makam. High Performance Metadata Integrity Protection in the WAFL Copy-on-Write File System. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 197–212, Santa Clara, CA, 2017. USENIX Association.

[113] Avinash Lakshman and Prashant Malik. Cassandra: A Decentralized Structured Storage System. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, April 2010.

[114] D. Lee, J. Choi, J. H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim. LRFU: A Spectrum of Policies That Subsumes the Least Recently Used and Least Frequently Used Policies. *IEEE Trans. Comput.*, 50(12):1352–1361, December 2001.

[115] Sangmin Lee, Rina Panigrahy, Vijayan Prabhakaran, Venugopalan Ramasubramanian, Kunal Talwar, Lincoln Uyeda, and Udi Wieder. Validating Heuristics for Virtual Machines Consolidation. Technical Report MSR-TR-2011-9, Microsoft Research, January 2011.

[116] Joshua LeVasseur, Volkmar Uhlig, Jan Stoess, and Stefan Götz. Unmodified Device Driver Reuse and Improved System Dependability via Virtual Machines. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation*, OSDI'04, Berkeley, CA, USA, 2004. USENIX Association.

[117] J. Liao and Y. Ishikawa. Partial Replication of Metadata to Achieve High Metadata Availability in Parallel File Systems. In *2012 41st International Conference on Parallel Processing*, pages 168–177, Sept 2012.

[118] libvmi.com. Libvmi: Virtual machine introspection library. `http://libvmi.com/` `https://github.com/libvmi/libvmi`, 2016.

[119] J. Liu, K. Zhou, L. Pang, Z. Wang, Y. Deng, and D. Feng. A Novel Cost-Effective Disk Scrubbing Scheme. In *2009 Fifth International Joint Conference on INC, IMS and IDC*, pages 686–691, Aug 2009.

[120] Ao Ma, Chris Dragga, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Marshall Kirk Mckusick. Ffsck: The Fast File-System Checker. *Trans. Storage*, 10(1):2:1–2:28, January 2014.

[121] Jean Jacques Maleval. History: Milestones in HDD Capacity. `https://www.storagenewsletter.com/2013/08/29/milestones-in-hdd-capacity/`, 2013.

[122] Sonam Mandal, Geoff Kuenning, Dongju Ok, Varun Shastry, Philip Shilane, Sun Zhen, Vasily Tarasov, and Erez Zadok. Using Hints to Improve Inline Block-layer Deduplication. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 315–322, Santa Clara, CA, February 2016. USENIX Association.

[123] P. D. Marinescu and G. Candea. LFI: A practical and general library-level fault injector. In *2009 IEEE/IFIP International Conference on Dependable Systems Networks*, pages 379–388, June 2009.

[124] Richard McDougall and Jim Mauro. FileBench. `http://filebench.sourceforge.net/`, 2005.

[125] Marshall Kirk McKusick. Running "Fsck" in the Background. In *Proceedings of the BSD Conference 2002 on BSD Conference*, BSDC'02, Berkeley, CA, USA, 2002. USENIX Association.

[126] Marshall Kirk Mckusick and T. J. Kowalski. Fsck - The UNIX File System Check Program, 1994.

[127] Nimrod Megiddo and Dharmendra S. Modha. ARC: A Self-Tuning, Low Overhead Replacement Cache. In *Proceedings of the 2Nd USENIX Conference on File and Storage Technologies*, FAST '03, pages 115–130, Berkeley, CA, USA, 2003. USENIX Association.

[128] Dirk Merkel. Docker: Lightweight Linux Containers for Consistent Development and Deployment. *Linux J.*, 2014(239), March 2014.

[129] M. Mesnier, G. R. Ganger, and E. Riedel. Object-based storage. *IEEE Communications Magazine*, 41(8):84–90, Aug 2003.

[130] M. Mesnier, E. Thereska, G. R. Ganger, D. Ellard, and M. Seltzer. File classification in self-* storage systems. In *International Conference on Autonomic Computing, 2004. Proceedings.*, pages 44–51, May 2004.

[131] Michael Mesnier, Feng Chen, Tian Luo, and Jason B. Akers. Differentiated storage services. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 57–70, New York, NY, USA, 2011. ACM.

[132] Dutch T. Meyer, Gitika Aggarwal, Brendan Cully, Geoffrey Lefebvre, Michael J. Feeley, Norman C. Hutchinson, and Andrew Warfield. Parallax: Virtual Disks for Virtual Machines. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, Eurosys '08, pages 41–54, New York, NY, USA, 2008. ACM.

[133] Justin Meza, Qiang Wu, Sanjev Kumar, and Onur Mutlu. A Large-Scale Study of Flash Memory Failures in the Field. In *Proceedings of the 2015 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '15, pages 177–190, New York, NY, USA, 2015. ACM.

[134] Ningfang Mi, A. Riska, E. Smirni, and E. Riedel. Enhancing data availability in disk drives through background activities. In *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*, pages 492–501, June 2008.

[135] Ethan L. Miller. Towards scalable benchmarks for mass storage systems. In *5th NASA Goddard Conference on Mass Storage Systems and Technologies*, 1996.

[136] Rich Miller. Ma.gnolia Data is Gone For Good. `http://www.datacenterknowledge.com/archives/2009/02/19/magnolia-data-is-gone-for-good/`, 2009.

[137] MongoDB Inc. MongoDB's 'repairDatabase' Tool and its Wrappers. `https://docs.mongodb.com/manual/reference/command/repairDatabase/`, 2017.

[138] Timothy Prickett Morgan. Cassandra Carves Storage Niche, Aims At Microservers. `https://www.nextplatform.com/2015/04/15/cassandra-carves-storage-niche-aims-at-microservers/`, 2017.

[139] James Myers. Data Integrity in Solid State Drives: What Supernovas Mean to You. `https://itpeernetwork.intel.com/data-integrity-in-solid-state-drives-what-supernovas-mean-to-you/`, 2014.

[140] Iyswarya Narayanan, Di Wang, Myeongjae Jeon, Bikash Sharma, Laura Caulfield, Anand Sivasubramaniam, Ben Cutler, Jie Liu, Badriddine Khessib, and Kushagra Vaid. SSD Failures in Datacenters: What? When? And Why? In *Proceedings of the 9th ACM International on Systems and Storage Conference*, SYSTOR '16, pages 7:1–7:11, New York, NY, USA, 2016. ACM.

[141] Suman Nath, Haifeng Yu, Phillip B. Gibbons, and Srinivasan Seshan. Subtleties in Tolerating Correlated Failures in Wide-area Storage Systems. In *Proceedings of the 3rd Conference on Networked Systems Design & Implementation - Volume 3*, NSDI'06, Berkeley, CA, USA, 2006. USENIX Association.

[142] Michael Nelson, Beng-Hong Lim, and Greg Hutchins. Fast Transparent Migration for Virtual Machines. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, Berkeley, CA, USA, 2005. USENIX Association.

[143] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. The LRU-K Page Replacement Algorithm for Database Disk Buffering. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, SIGMOD '93, pages 297–306, New York, NY, USA, 1993. ACM.

[144] Alina Oprea and Ari Juels. A Clean-slate Look at Disk Scrubbing. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies*, FAST'10, Berkeley, CA, USA, 2010. USENIX Association.

[145] Oracle Corporation. MySQL White Papers, 2016. `https://www.mysql.com/why-mysql/white-papers/`.

[146] Bernd Panzer-Steindel. Data integrity. *CERN/IT*, 2007.

[147] David Patterson. For Better or Worse, Benchmarks Shape a Field: Technical Perspective. *Commun. ACM*, 55(7), July 2012.

[148] David Patterson, Aaron Brown, Pete Broadwell, George Candea, Mike Chen, James Cutler, Patricia Enriquez, Armando Fox, Matthew Merzbacher, et al. Recovery-oriented computing (ROC): Motivation, definition, techniques, and case studies. Technical report, 2002.

[149] David A. Patterson, Garth Gibson, and Randy H. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, SIGMOD '88, pages 109–116, New York, NY, USA, 1988. ACM.

[150] B. D. Payne, M. D. P. D. A. Carbone, and W. Lee. Secure and Flexible Monitoring of Virtual Machines. In *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*, pages 385–397, Dec 2007.

[151] Ben Pfaff, Tal Garfinkel, and Mendel Rosenblum. Virtualization Aware File Systems: Getting Beyond the Limitations of Virtual Disks. In *Proceedings of the 3rd Conference on Networked Systems Design and Implementation - Volume 3*, NSDI'06, Berkeley, CA, USA, 2006. USENIX Association.

[152] Jonas Pfoh, Christian Schneider, and Claudia Eckert. *Nitro: Hardware-Based System Call Tracing for Virtual Machines*, pages 96–112. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[153] Eduardo Pinheiro, Wolf-Dietrich Weber, and Luiz André Barroso. Failure Trends in a Large Disk Drive Population. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies*, FAST '07, Berkeley, CA, USA, 2007. USENIX Association.

[154] Florentina I. Popovici, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Robust, Portable I/O Scheduling with the Disk Mimic. pages 297–310.

[155] Vijayan Prabhakaran, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Analysis and Evolution of Journaling File Systems. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, Berkeley, CA, USA, 2005. USENIX Association.

[156] Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. IRON File Systems. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, SOSP '05, pages 206–220, New York, NY, USA, 2005. ACM.

[157] David Reinsel, John Gantz, and John Rydning. Total WW Data to Reach 163ZB by 2025. `https://www.storagenewsletter.com/2017/04/05/total-ww-data-to-reach-163-zettabytes-by-2025-idc/`, 2017.

[158] Erik Riedel, Mahesh Kallahalla, and Ram Swaminathan. A Framework for Evaluating Storage System Security. pages 14–29.

[159] Ohad Rodeh, Josef Bacik, and Chris Mason. BTRFS: The Linux B-Tree Filesystem. *Trans. Storage*, 9(3):9:1–9:32, August 2013.

[160] Ohad Rodeh and Avi Teperman. zFS: A Scalable Distributed File System Using Object Disks. In *Proceedings of the 20 th IEEE/11 th NASA Goddard Conference on Mass Storage Systems and Technologies (MSS'03)*, MSS '03, Washington, DC, USA, 2003. IEEE Computer Society.

[161] Drew Roselli, Jacob R. Lorch, and Thomas E. Anderson. A Comparison of File System Workloads. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '00, Berkeley, CA, USA, 2000. USENIX Association.

[162] M. Rosenblum and T. Garfinkel. Virtual Machine Monitors: Current Technology and Future Trends. *Computer*, pages 39–47, 2005.

[163] Mendel Rosenblum. The Reincarnation of Virtual Machines. *Queue*, 2(5), 2004.

[164] Mendel Rosenblum, Edouard Bugnion, Scott Devine, and Stephen A. Herrod. Using the SimOS Machine Simulator to Study Complex Computer Systems. *ACM Trans. Model. Comput. Simul.*, 7(1):78–103, January 1997.

[165] Paul Rubin. Shipwrecked! A MongoDB Data Recovery Tale. `https://www.compose.com/articles/shipwrecked-a-mongodb-data-recovery-tale/`, 2013.

[166] Chris Ruemmler and John Wilkes. An Introduction to Disk Drive Modeling. 27(3):17–28, March 1994.

[167] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end Arguments in System Design. *ACM Trans. Comput. Syst.*, 2(4):277–288, November 1984.

[168] J. Schaffner, D. Jacobs, B. Eckart, J. Brunnert, and A. Zeier. Towards enterprise software as a service in the cloud. In *2010 IEEE 26th International Conference on Data Engineering Workshops (ICDEW 2010)*, pages 52–59, March 2010.

[169] J. Schindler and G. Ganger. Automated disk drive characterization. Technical Report CMU-CS-99-176, Carnegie Mellon University, November 1999.

[170] Jiri Schindler. I/O Characteristics of NoSQL Databases. *Proc. VLDB Endow.*, 5(12):2020–2021, August 2012.

[171] Jiri Schindler, John Linwood Griffin, Christopher R. Lumb, and Gregory R. Ganger. Track-Aligned Extents: Matching Access Patterns to Disk Drive Characteristics. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, FAST '02, Berkeley, CA, USA, 2002. USENIX Association.

[172] Frank Schmuck and Roger Haskin. Gpfs: A shared-disk file system for large computing clusters. In *FAST '02: Proceedings of the 1st USENIX Conference on File and Storage Technologies*, page 19, Berkeley, CA, USA, 2002. USENIX Association.

[173] Bianca Schroeder, Sotirios Damouras, and Phillipa Gill. Understanding Latent Sector Errors and How to Protect Against Them. *Trans. Storage*, 6(3):9:1–9:23, September 2010.

[174] Bianca Schroeder and Garth A. Gibson. Disk Failures in the Real World: What Does an MTTF of 1,000,000 Hours Mean to You? In *Proceedings of the 5th USENIX Conference on File and Storage Technologies*, FAST '07, Berkeley, CA, USA, 2007. USENIX Association.

[175] Bianca Schroeder, Raghav Lagisetty, and Arif Merchant. Flash Reliability in Production: The Expected and the Unexpected. In *Proceedings of the 14th Usenix Conference on File and Storage Technologies*, FAST'16, pages 67–80, Berkeley, CA, USA, 2016. USENIX Association.

[176] Bianca Schroeder, Eduardo Pinheiro, and Wolf-Dietrich Weber. DRAM Errors in the Wild: A Large-scale Field Study. In *Proceedings of the Eleventh International Joint Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '09, pages 193–204, New York, NY, USA, 2009. ACM.

[177] T. J. E. Schwarz, Qin Xin, E. L. Miller, D. D. E. Long, A. Hospodor, and S. Ng. Disk scrubbing in large archival storage systems. In *Modeling, Analysis, and Simulation of Computer and Telecommunications Systems, 2004. (MASCOTS 2004). Proceedings. The IEEE Computer Society's 12th Annual International Symposium on*, pages 409–418, Oct 2004.

[178] S. Shah and J. G. Elerath. Disk drive vintage and its effect on reliability. In *Annual Symposium Reliability and Maintainability, 2004 - RAMS*, pages 163–167, Jan 2004.

[179] S. Shah and J. G. Elerath. Reliability analysis of disk drive failure mechanisms. In *Annual Reliability and Maintainability Symposium, 2005. Proceedings.*, pages 226–231, Jan 2005.

[180] Mohammad Shamma, Dutch T. Meyer, Jake Wires, Maria Ivanova, Norman C. Hutchinson, and Andrew Warfield. Capo: Recapitulating Storage for Virtual Desktops. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies*, FAST'11, Berkeley, CA, USA, 2011. USENIX Association.

[181] Ross Shaull, Liuba Shrira, and Hao Xu. Skippy: A New Snapshot Indexing Method for Time Travel in the Storage Manager. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 637–648, New York, NY, USA, 2008. ACM.

[182] L. Shrira and H. Xu. SNAP: efficient snapshots for back-in-time execution. In *21st International Conference on Data Engineering (ICDE'05)*, pages 434–445, April 2005.

[183] Liuba Shrira and Hao Xu. Thresher: An Efficient Storage Manager for Copy-on-write Snapshots. In *Proceedings of the Annual Conference on USENIX '06 Annual Technical Conference*, ATEC '06, Berkeley, CA, USA, 2006. USENIX Association.

[184] Elizabeth Shriver. *Performance modeling for realistic storage devices*. PhD thesis, New York, NY, USA, 1997.

[185] Gopalan Sivathanu, Swaminathan Sundararaman, and Erez Zadok. Type-Safe Disks. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 15–28, Berkeley, CA, USA, 2006. USENIX Association.

[186] Muthian Sivathanu, Lakshmi N. Bairavasundaram, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Life or Death at Block-level. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design and Implementation - Volume 6*, OSDI'04, Berkeley, CA, USA, 2004. USENIX Association.

[187] Muthian Sivathanu, Lakshmi N. Bairavasundaram, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Database-aware Semantically-smart Storage. In *Proceedings of the 4th Conference on USENIX Conference on File and Storage Technologies - Volume 4*, FAST'05, Berkeley, CA, USA, 2005. USENIX Association.

[188] Muthian Sivathanu, Vijayan Prabhakaran, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Improving Storage System Availability with D-GRAID. *Trans. Storage*, 1(2):133–170, May 2005.

[189] Muthian Sivathanu, Vijayan Prabhakaran, Florentina I. Popovici, Timothy E. Denehy, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Semantically-Smart Disk Systems. In *Proceedings of the 2Nd USENIX Conference on File and Storage Technologies*, FAST '03, pages 73–88, Berkeley, CA, USA, 2003. USENIX Association.

[190] David M. Smith. The Cost of Lost Data. `https://gbr.pepperdine.edu/2010/08/the-cost-of-lost-data`, 2010.

[191] Ivan Smith. Cost of Hard Drive Storage Space. `http://ns1758.ca/winch/winchest.html`, 2013.

[192] sortbenchmark.org. GraySort Benchmark. `http://sortbenchmark.org/`.

[193] Gokul Soundararajan, Vijayan Prabhakaran, Mahesh Balakrishnan, and Ted Wobber. Extending SSD Lifetimes with Disk-based Write Caches. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies*, FAST'10, Berkeley, CA, USA, 2010. USENIX Association.

[194] Vilas Sridharan, Nathan DeBardeleben, Sean Blanchard, Kurt B. Ferreira, Jon Stearley, John Shalf, and Sudhanva Gurumurthi. Memory Errors in Modern Systems: The Good, The Bad, and The Ugly. *SIGARCH Comput. Archit. News*, 43(1):297–310, March 2015.

[195] Sriram Subramanian, Yupu Zhang, Rajiv Vaidyanathan, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Jeffrey F. Naughton. Impact of Disk Corruption on Open-Source DBMS. In *Proceedings of the 26th International Conference on Data Engineering*, Long Beach, California, March 2010.

[196] Standard Performance Evaluation Corporation. SPECmail2009 Benchmark. `http://www.spec.org/mail2009/`.

[197] Michael Stonebraker. The Design of the POSTGRES Storage System. In *Proceedings of the 13th International Conference on Very Large Data Bases*, VLDB '87, pages 289–300, San Francisco, CA, USA, 1987. Morgan Kaufmann Publishers Inc.

[198] Mark W. Storer, Kevin Greenan, Darrell D.E. Long, and Ethan L. Miller. Secure Data Deduplication. In *Proceedings of the 4th ACM International Workshop on Storage Security and Survivability*, StorageSS '08, pages 1–10, New York, NY, USA, 2008. ACM.

[199] V. Sundaram and P. Shenoy. A Practical Learning-based Approach for Dynamic Storage Bandwidth Allocation. Technical report, Amherst, MA, USA, 2002.

[200] Adan Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. Scalability in the XFS File System.

[201] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the Reliability of Commodity Operating Systems. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 207–222, New York, NY, USA, 2003. ACM.

[202] Technical Committee T10. SCSI Storage Interfaces. `http://t10.org/`, 2017.

[203] Nisha Talagala, Remzi H. Arpaci-Dusseau, and Dave Patterson. Microbenchmark-based Extraction of Local and Global Disk Characteristics. Technical Report CSD-99-1063, University of California, Berkeley, 1999.

[204] Nisha Talagala and David Patterson. An Analysis of Error Behaviour in a Large Storage System. In *The IEEE Workshop on Fault Tolerance in Parallel and Distributed Systems*, San Juan, Puerto Rico, April 1999.

[205] Vasily Tarasov, Deepak Jain, Dean Hildebrand, Renu Tewari, Geoff Kuenning, and Erez Zadok. Improving I/O Performance Using Virtual Disk Introspection. In *Presented as part of the 5th USENIX Workshop on Hot Topics in Storage and File Systems*, Berkeley, CA, 2013. USENIX.

[206] Vasily Tarasov, Sonam Mandal, Philip Shilane, Deepak Jain, Geoff Kuenning, Karthikeyani Palanisami, Sagar Trehan, and Erez Zadok. Dmdedup: Device Mapper Target for Data Deduplication.

[207] TheRegister. Admin downs entire Joyent data center. `http://www.theregister.co.uk/2014/05/28/joyent_cloud_down/`.

[208] Eno Thereska, Hitesh Ballani, Greg O'Shea, Thomas Karagiannis, Antony Rowstron, Tom Talpey, Richard Black, and Timothy Zhu. IOFlow: A Software-defined Storage Architecture. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 182–196, New York, NY, USA, 2013. ACM.

[209] Avishay Traeger and Erez Zadok. How to cheat at benchmarking. In *USENIX FAST Birds of a feather session*, San Francisco, CA, February 2009.

[210] Transaction Processing Council. TPC Benchmark H Standard Specification, Revision 2.17.1. `http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-h_v2.17.1.pdf`, 2014.

[211] Vassilis J. Tsotras and Nickolas Kangelaris. The snapshot index: An i/o-optimal access method for timeslice queries. *Information Systems*, 20(3):237 – 260, 1995.

[212] Stephen C. Tweedie. Journaling the Linux ext2fs File System. In *The Fourth Annual Linux Expo*, Durham, North Carolina, May 1998.

[213] Karl Ulrich. *Fundamentals of Product Modularity*, pages 219–231. Springer Netherlands, Dordrecht, 1994.

[214] Sandeep Uttamchandani, Kaladhar Voruganti, Sudarshan Srinivasan, John Palmer, and David Pease. Polus: Growing Storage QoS Management Beyond a "4-Year Old Kid". In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, FAST '04, pages 31–44, Berkeley, CA, USA, 2004. USENIX Association.

[215] VMware Inc. VMware VMFS product datasheet. `http://www.vmware.com/pdf/vmfs_datasheet.pdf`.

[216] Werner Vogels. Beyond Server Consolidation. *Queue*, 6(1):20–26, 2008.

[217] Carl Waldspurger and Mendel Rosenblum. I/O Virtualization. *Commun. ACM*, 55(1):66–73, January 2012.

[218] Yang Wang, Lorenzo Alvisi, and Mike Dahlin. Gnothi: Separating Data and Metadata for Efficient and Available Storage Replication. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, Berkeley, CA, USA, 2012. USENIX Association.

[219] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. Scale and Performance in the Denali Isolation Kernel. *SIGOPS Oper. Syst. Rev.*, 36(SI):195–209, December 2002.

[220] M. Wittle and Bruce E. Keith. LADDIS: The Next Generation in NFS File Server Benchmarking. In *USENIX Summer*, 1993.

[221] 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, Dec 2010.

[222] Bruce L. Worthington, Gregory R. Ganger, Yale N. Patt, and John Wilkes. On-Line Extraction of SCSI Disk Drive Parameters. Technical Report CSE-TR-323-96, Carnegie Mellon University, 19 1996.

[223] Michael Wu and Willy Zwaenepoel. eNVy: A Non-volatile, Main Memory Storage System. *SIGPLAN Not.*, 29(11):86–97, November 1994.

[224] Jian Xu and Steven Swanson. NOVA: A log-structured file system for hybrid volatile/non-volatile main memories. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 323–338, Santa Clara, CA, 2016. USENIX Association.

[225] Chaitanya Yalamanchili, Kiron Vijayasankar, Erez Zadok, and Gopalan Sivathanu. DHIS: Discriminating Hierarchical Storage. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, SYSTOR '09, pages 9:1–9:12, New York, NY, USA, 2009. ACM.

[226] Junfeng Yang, Can Sar, and Dawson Engler. EXPLODE: A Lightweight, General System for Finding Serious Storage System Errors. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, OSDI '06, Berkeley, CA, USA, 2006. USENIX Association.

[227] YCombinator. Joyent us-east-1 rebooted due to operator error. `https://news.ycombinator.com/item?id=7806972`.

[228] Erez Zadok. File and storage systems benchmarking workshop. UC Santa Cruz, CA, May 2008.

[229] Yiying Zhang, Leo Prasath Arulraj, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. De-indirection for Flash-based SSDs with Nameless Writes. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, FAST'12, Berkeley, CA, USA, 2012. USENIX Association.

[230] Yupu Zhang, Abhishek Rajimwale, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. End-to-end Data Integrity for File Systems: A ZFS Case Study. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies*, FAST'10, Berkeley, CA, USA, 2010. USENIX Association.

[231] Ping Zhou, Bo Zhao, Jun Yang, and Youtao Zhang. A Durable and Energy Efficient Main Memory Using Phase Change Memory Technology. *SIGARCH Comput. Archit. News*, 37(3):14–23, June 2009.

[232] Yuanyuan Zhou, James Philbin, and Kai Li. The Multi-Queue Replacement Algorithm for Second Level Buffer Caches. In *Proceedings of the General Track: 2001 USENIX Annual Technical Conference*, pages 91–104, Berkeley, CA, USA, 2001. USENIX Association.

[233] Benjamin Zhu, Kai Li, and Hugo Patterson. Avoiding the Disk Bottleneck in the Data Domain Deduplication File System. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, FAST'08, pages 18:1–18:14, Berkeley, CA, USA, 2008. USENIX Association.