Understanding and Improving the Performance of Read Operations Across the Storage Stack

María Fernanda Borge Chávez

Advisor: Prof. Willy Zwaenepoel

School of Computer Science

The University of Sydney

A thesis submitted to The University of Sydney in fulfilment of the requirements for the degree of *Masters of Philosophy*

December 2019

Statement of Originality

This is to certify that to the best of my knowledge, the content of this thesis is my own work. This thesis has not been submitted for any degree or other purposes.

I certify that the intellectual content of this thesis is the product of my own work and that all the assistance received in preparing this thesis and sources have been acknowledged.

Name:

Signature: **Date**:

Statement of Attribution

The research presented in this thesis was conducted under the supervision of Prof. Willy Zwaenepoel at The University of Sydney in Australia. The main results presented in this thesis were first introduced in the following publications:

1. Pulkit Misra, María F. Borge, Íñigo Goiri, Alvin R. Lebeck, Willy Zwaenepoel, Ricardo Bianchini. *Managing Tail Latency in Datacenter-Scale FileSystems Under Production Constraints*. 14th European Conference on Computer Systems (EuroSys), Dresden, Germany, 2019 [77]. (Chapter 4)

2. María F. Borge, Florin Dinu, Willy Zwaenepoel. *Understanding and taming SSD read performance variability: HDFS case study*. Technical report (arXiv:1903.09347), 2019 [29]. (Chapter 3)

Chapter 4 focuses on smart hedging, the contribution of the author of this thesis. The performance characterization and evaluation were performed together with co-authors. The technique to mitigate tail latency in write operations, fail fast, is not a topic of this thesis.

As supervisor for the candidature upon which this thesis is based, I can confirm that the authorship attribution statements above are correct.

Name: Willy Zwaenepoel

Signature: **Date**:

As the first author of the pulication *Managing Tail Latency in Datacenter-Scale FileSystems Under Production Constraints*, presented at the 14th European Conference on Computer Systems (EuroSys) in Dresden, Germany on 2019, I can confirm that the autorship attribution statements above are correct. I grant permission to include the published material in this thesis.

Name: Pulkit A. Misra

Signature: **Date**:

In addition to the statements above, in cases where I am not the first author of a published item, permission to include the published material has been granted by the first author.

Name: María Fernada Borge Chávez

Signature: **Date**:

Abstract

We live in a data-driven era, large amounts of data are generated and collected every day. Storage systems are the backbone of this era, as they store and retrieve data. To cope with increasing data demands (*e.g.*, diversity, scalability), storage systems are experiencing changes across the stack. As other computer systems, storage systems rely on layering and modularity, to allow rapid development. Unfortunately, this can hinder performance clarity and introduce degradations (*e.g.*, tail latency), due to unexpected interactions between components of the stack.

In this thesis, we first perform a study to understand the behavior across different layers of the storage stack. We focus on sequential read workloads, a common I/O pattern in distributed file systems (*e.g.*, HDFS, GFS). We analyze the interaction between read workloads, local file systems (*i.e.*, ext4), and storage media (*i.e.*, SSDs). We perform the same experiment over different periods of time (*e.g.*, file lifetime). We uncover 3 slowdowns, all of which occur in the lower layers. When combined, these slowdowns can degrade throughput by 30%. We find that increased parallelism on the local file system mitigates these slowdowns, showing the need for adaptability in storage stacks.

Given the fact that performance instabilities can occur at any layer of the stack, it is important that upper-layer systems are able to react. We propose *smart hedging*, a novel technique to manage high-percentile (tail) latency variations in read operations. Smart hedging considers production challenges, such as massive scalability, heterogeneity, and ease of deployment and maintainability. Our technique establishes a dynamic threshold by tracking latencies on the client-side. If a read operation exceeds the threshold, a new hedged request is issued, in an exponential back-off manner. We implement our technique in HDFS and evaluate it on

70k servers in 3 datacenters. Our technique reduces average tail latency, without generating excessive system load.

Acknowledgements

First and foremost, I would like to thank the people whose collaborations made this thesis possible. I am grateful to my advisor, Willy Zwanapoel. Willy provided the advice and resources to work on exciting research topics. I would also like to thank Ricardo Bianchini and Íñigo Goiri. Both of them mentored me during an internship at Microsoft Research Redmond. Ricardo and Íñigo shared their research experience, inspired different research directions, provided honest advice. From them, I learned how to apply academic research into an industrial lab and product. I am also thankful to Florin Dinu. Florin showed me how to structure my thoughts, structure experiments, provided thoughtful and through feedback on presentation and writing skills.

I am also grateful to Karan Gupta and Vinayak Khot, my mentors during an internship at Nutanix San Jose. Karan and Vinayak introduced me to a dynamic industrial environment. Moreover, they showed me the immediate impact that research has on a product.

I am thankful to the LABOS members: Florin, Kristina, Jasmina, Oana, Calin, Pamela, Laurent, Baptiste, and Diego. Thank you for all the conversations, feedback, experiences shared, help, and advice. I would also like to thank to Florin Dinu and Alan Fekete, for their feedback on the writing of this thesis.

A big thank you to my friends from Mexico, Daniel, Edgar Karam, Jorge, María José, Mónica, Raúl and Sonia; and my friends from Switzerland: Adrien and Hussein, Thank you all for always being there for me, no matter the time zone! For finding time to talk and listen to me when needed the most; for putting difficult situations under perspective; for always making me smile and laugh; for supporting me on every decision.

I would also like to thank my family. Specially my mom, Lupita, for all her love and support; for always believing and me; for always being there for me, even though we were physically far away.

Last but not least, I would also like to thank Clément. He has given me his unconditional love and unwavering support, no matter the country, the continent, or the time zone.

Table of contents

List of figures

List of tables

Chapter 1

Introduction

We are experiencing a massive data growth. In 2017, IBM revealed that 90% of the world's data had been created in 2015 and 2016 alone [56]. Seagate projects that the global data size will reach 175 Zettabytes by 2025, 10 times more than in 2016 [85, 86]. Data powers different services that enhance our every day life. Ride-sharing services and autonomous cars rely on data for cost-efficient route planning. Robots that assist the elderly and aid in education. Smart home devices (*e.g.*, smart locks, smart plugs, virtual assistants) are present in many homes. These data intensive services, among many others, do not only analyze existing data, but are constantly generating it.

In an environment of endless data collection and analysis, storage is a critical piece of infrastructure. The increase in demand, both in volume and types of data, has unleashed a complete revolution of the storage stack [20]. At the bottom of the stack, new storage media are introduced (*e.g.*, PCIe SSDs, shingled-mangetic recording (SMR) disks). New local file systems are designed to better manage new storage technologies [64, 59, 66], while existing ones evolve and adapt [19, 74]. On the higher levels of the stack, new data service platforms, ranging from key-value stores (*e.g.*, RocksDB [46], HBase [26], Cassandra [65]) to distributed file systems (*e.g.*, GFS [47], HDFS [89], Cosmos [38]) have been developed to manage different types of data.

Most of these storage systems run on cloud platforms [86] (*e.g.*, AWS [2], GCP [6], Azure [8], Nutanix [9]) and are hosted in datacenters [4] with thousands of servers [21, 35, 76, 77, 82, 115]. The rapid data explosion and large scale have impacted the design of current and next generation storage systems.

As a result, large body of storage systems platforms rely on layered architectures, rather than on building end-to-end platforms from the ground up. Layering relieves system engineers from the complex tasks of designing and deploying large-scale systems from the ground up. For instance, HBase [26], a distributed key-value store, relies on the Hadoop Distributed File System (HDFS) [89], a user-space distributed file system, to provide fault-tolerance mechanisms. In turn, HDFS leverages local file system's interfaces to simplify the interaction with underlaying block devices and provide portability. However, layering can hinder performance understanding and efficiency [53, 87, 99].

Furthermore, in large-scale environments (*e.g.*, large datacenters), even carefully designed and tested systems can suffer unexpected performance degradations [52, 84]. In today's data intensive era, users and applications demand low response times. Consequently, unpredictable high-percentile (tail) performance variations are costly and become intolerable. Several studies show that high response times can drive users away and decrease revenues [30, 91]. However, identifying the sources of these performance variations is not trivial, since they can have different root causes, such as component failures, replication overhead, resource contention, software bugs, among many others.

Additionally, these variations can simultaneously occur at any layer of the stack [53, 67, 87]. The problem becomes more challenging when systems run on harvested resources [76, 112, 115]. Resource-harvesting datacenters improve resource utilization via the co-location of batch jobs (*e.g.*, data analytics, machine learning) and latency-sensitive services (*e.g.*, search engines). In these settings, performance isolation mechanisms [43, 48, 57, 70, 73, 101, 109, 111, 112, 115] manage (*i.e.*, throttle or deny) the resources of batch jobs according to the needs of latency-sensitive services.

Consequently, as the storage stack evolves, both in terms of software and hardware, understanding the interaction between different components becomes indispensable to reduce performance variations. Moreover, as performance instabilities vary depending on different system characteristics, such as scale, it becomes relevant to study widely used workloads and behaviors across different scenarios.

1.1 Contributions

In this thesis, we focus on *understanding and improving the performance of read workloads in storage systems*. More specifically, we target large read operations ($>$ 1MB), a common workload in distributed file system [38, 47, 89]. The complexity of systems, in both design (layering) and scale, can obscure performance understanding and introduce unexpected performance variations. The goal of this work is to understand and improve performance in both situations. To this end, we first study the interaction between different components of the storage stack (*e.g.*, application, local file system, block layer, storage device), as well as their impact on performance variability. Since performance degradations can occur at any component of the storage stack, it is important to provide applications with mechanisms to react. The second part of this thesis studies the performance of high-percentile (tail) latency, with a focus on the user-space application. In particular, we examine the behavior of distributed file systems in large-scale (tens of thousands of servers) production settings. We provide a brief overview of the components of this thesis in the paragraphs below.

1.1.1 Understanding the Performance of Read Operations in SSDs

The first part of this thesis seeks to understand performance instabilities caused by the interactions between different layers of the storage stack. Our study focuses on large sequential read operations, a common workload in distributed file systems [38, 47, 89]. We

decide to solely investigate read workloads because they involve less complex interactions, both with software and hardware, than write operations. Additionally, most of the insights observe in read workloads will typically affect write workloads, paving the way for insights on write workloads.

In this work, we particularly study the interactions between large sequential read workloads with ext4 [74] as a local file system, and three SATA SSDs (from different manufacturers) as the storage media. Our study reveals that the majority of performance degradation come from components in the lower layers of the storage stack. In particular, they are caused by the interaction between the local file system and SSDs.

We expose and investigate three slowdowns that negatively impact read throughput in large sequential read operations. We refer to these slowdowns as intrinsic, temporal, and permanent. First, the intrinsic slowdown, which reduces read throughput for a (variable) short amount of time (seconds). The intrinsic slowdown takes place at the start of every new ext4 extent read only tested on ext4, but our observations might be applicable to other extent based local file systems. Second, the temporal slowdown, which affects read operations at medium time scales (minutes to hours). In the temporal slowdown, storage device tail latencies increase periodically and temporarily, producing throughput degradation. The effect of this slowdown is very similar to reported effects of write-triggered SSD garbage collection [90, 108]. However, we find that this slowdown occurs on read-only workloads. Finally, the permanent slowdown, which influences read operations permanently and develops in long time scales (days to weeks). After a long period of time (of file lifetime), read throughput permanently decreases and never recovers. It is important to note, that this slowdown affects files individually, and it is not caused by a single drive-wide malfunction. In comparison to temporal slowdown, all requests in permanent slowdown show an increase in storage device latency, not only the tail. Via experimentation, we show that each slowdown can individually degrade throughput (at least) between 10% to 15%. When all three slowdowns combine, we observe a throughput loss of up to 30%.

We find that these slowdowns can be masked via increased parallelism on the lower layers. However, HDFS, as well as other storage systems, cannot trigger this increased parallelism by default. Our results point to a need for developing more adaptable storage stacks. We believe that understanding which interactions cause performance instability, as well as its temporal characteristics can provide helpful advice towards the future design of different layers in the storage stack.

1.1.2 Reducing Read Tail Latency in Distributed File Systems

In the second part of this thesis, we focus on reducing tail latency for read operations. Specifically, we focus on large-scale (tens of thousands of servers) distributed file systems with real production-constraints. Tail latency management in distributed systems has been addressed by several prior works, from data analytics frameworks [22, 23, 92, 113], distributed storage systems [68, 93], to systems that enforce service-level objectives for compute or storage [62, 71, 97, 102, 104, 105, 116]. Unfortunately, these works do not account for some important constraints that are present in *real production systems*. These constraints are exacerbated in resource harvesting datacenters in which batch workloads are co-located with latency-sensitive services [101, 112, 115]. The development of effective solutions in this context becomes critical, specially as the majority of the world's data moves to the the cloud [4, 86].

Some of the constraints present in production datacenters include: resource-harvesting, static heterogeneity (*i.e.*, different server hardware), dynamic heterogeneity (*i.e.*, produced by performance isolation mechanisms), scalability (*e.g.*, centralized components), and maintainability.

In this thesis, we address the challenging scenario where the distributed file system only stores data for the batch workloads, but the latency-sensitive services have full priority over

the shared resources. We propose *smart hedging*, a production-complaint technique to reduce read operation tail latency in distributed file systems.

Smart hedging is oblivious to the source of variations, relies on client-side tracking, and exploits already existing replication and fault-tolerance mechanisms in distributed file systems. Our technique monitors, in the client, server-side performance reporting on a per-packet basis. Smart hedging triggers a "hedge" [41] (duplicate) request when performance starts to degrade. Excessive hedging may cause server overload, to prevent this situation we hedge adaptively and (exponentially) back-off from a server that does not complete a hedge before the original request.

We implement smart hedging as part of a client library in HDFS, which we call "CurtailHDFS"[1]. Our evaluation uses synthetic workloads on 4,000 servers; and production workloads on 70,000 servers across 3 Microsoft's datacenters. We reduce the average read latency by 1.4× compared to state-of-the-practice hedging, without server overloading. This is a significant improvement, specifically given the limited scope of our changes, which work on top of existing techniques (*e.g.*, speculative task execution). Even though we evaluate our techniques on highly heterogeneous resource-harvesting datacenters, we believe that they are applicable in other contexts as well.

1.1.3 Summary of Contributions

To summarize, this thesis makes the following contributions:

- We identify and analyze the intrinsic slowdown. We show that it affects read operations at the start of every new ext4 extent, for a variable amount of time.

[1] CurtailHDFS [77] reduces tail latency operations for both read and write operations. Smart hedging, the technique to mitigate tail latency in read operations, is the contribution of this thesis. The technique that CurtailHDFS uses to reduce tail latency in write operations is not a topic of discussion in this thesis.

- We identify and analyze the temporal slowdown. We show how it affects read operations in a periodic and temporal manner.
- We identify and analyze the permanent slowdown. We show read operations can be permanently affected on a per file basis, rather than via a full drive failure.
- We characterize the tail latency of batch jobs running on production HDFS in resource-harvesting datacenters.
- We propose smart hedging, a client-side technique to manage read operation tail latency in distributed file systems, and implemented it in HDFS.
- We evaluate our technique with both synthetic and real production workloads, and show that the average tail latency decreases without generating extensive system load.

1.2 Outline

The rest of this thesis is structured as follows. Chapter 2 describes relevant background on storage systems. Chapter 3 presents an study on the performance of sequential read operations in SSDs, it presents some temporal performance degradations. Chapter 4 details the motivation, challenges, design, and evaluation of smart hedging, our technique to mitigate read tail latencies in distributed file systems. Chapter 5 proposes future research directions. Finally, chapter 6 concludes.

Chapter 2

Background

This chapter describes the necessary background, architecture and access pattern, of the Hadoop Distributed File System (HDFS) related to this thesis.

2.1 Hadoop Distributed File System

2.1.1 Architecture

The Hadoop Distributed File System (HDFS) [12, 89] is a popular open-source system that has been used by companies like Yahoo [89], Facebook [53], Microsoft [76, 115], Twitter [7], and Spotify [78]. HDFS is based on the Google File System (GFS) [47]. To provide portability, HDFS is implemented in Java as a user-space file system. Thus, HDFS relies on the underlying local file system (*e.g.*, ext4 [74], XFS [94]) as backend storage. An HDFS file is composed of several blocks, each block is stored as one separate file in the local file system. Blocks are typically large files, 256MB is a common size in real world deployments [76, 115].

The HDFS design implements two services, a metadata manager service and a per-server block storage. The NameNode is the primary metadata manager, it is a centralized component

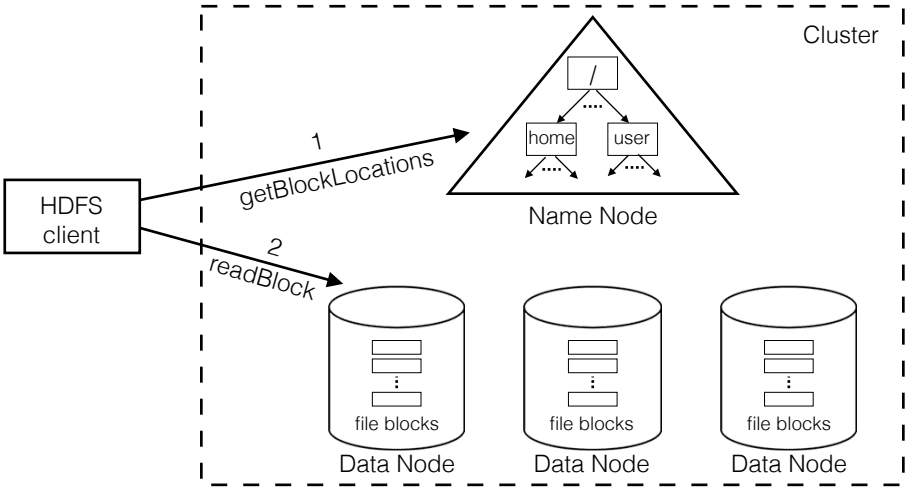


Fig. 2.1 Read a block in HDFS.

that operates on a single server. This service manages the namespace and maps HDFS files to their corresponding blocks. The NameNode does not maintain HDFS data, but rather keeps a mapping between HDFS file name and a list of DataNode(s) on which the blocks are stored. HDFS clients contact the NameNode to perform regular file system operations (*e.g.*, open, close, rename, delete). The DataNode is the per-server block storage, and is implemented by every other server in the cluster. Each DataNode stores HDFS blocks in the server's local file system. The DataNodes create or destroy blocks at the request of the NameNode, which handle client's requests. Although the NameNode manages the namespace, clients communicate directly with DataNodes in order to read or write data at the HDFS block level. Next, we describe read operations on an HDFS file.

Reads. Figure 2.1 shows the steps in reading a block. First, the client asks the NameNode for the block's locations. Then, the NameNode returns a list of DataNodes that store the block replicas; the list is sorted based on proximity to the client. Next, the client establishes a connection with the first DataNode on the list and reads one packet (1MB) at a time. In case of errors (*e.g.*, DataNode failure or data corruption), the client attempts to read the remaining packets from the next DataNode on the list. HDFS read operations access files

with a streaming access patterns. Each read operation performs several sequential iterations, during which it sequentially retrieves large data segments (*e.g.*, hundreds of kBs).

2.1.2 Access Pattern

We now summarize the main characteristics of the HDFS read access pattern since this pattern is central to our work.

- **Single-threaded.** One request from a compute task is handled by a single worker thread in the DataNode.
- **Large files.** HDFS blocks are large files (*e.g.*, 128MB, 256MB). Each HDFS block is a separate file in the local file system.
- **Sequential access.** The HDFS reads access data sequentially for performance reasons.
- **Buffered I/O.** HDFS uses buffered I/O in the DataNode (via `sendfile()` or `read()` system calls).

Chapter 3

Understanding Performance of Read Operations in SSDs

3.1 Motivation and Overview

Layering is a popular approach to build big data processing systems and it is particularly central to the design of storage stacks for big data systems. Even the simplest designs involve lots of layers: a data-processing system (*e.g.*, Hadoop [25], Spark [27]) processes data stored in a distributed file system (*e.g.*, HDFS [12, 89], GFS [47]), which in turn relies on an OS running a local file system (*e.g.*, ext4 [74]) which, finally, manages data on disks. Oftentimes, additional layers are used such as big data stores (*e.g.*, BigTable [39], HBase [26]) that are layered on top of the distributed file systems. Such layered architectures are in part responsible for the massive success and diversity of data processing systems today because they encourage rapid development by reusing existing functionality.

Regardless of the specific layers in a storage stack the goal of fully utilizing the capabilities of the underlying storage remains equally important. Layering makes this goal more challenging to achieve, because each layer can introduce potential performance bottlenecks as well as sources of performance variability. Performance bottlenecks in upper layers can mask suboptimal performance behavior in lower layers. Moreover, layering significantly complicates

root-cause analysis of performance problems and this can lead to blame being assigned to the wrong layer due to lack of information.

In this chapter we perform a deep dive into the performance of a critical layer in today's big data storage stacks, namely the Hadoop Distributed File System (HDFS). We particularly focus on the influence that other layers have on HDFS' ability to extract the maximum throughput that the underlying storage is capable of providing. We analyze and discover a number of software and hardware bottlenecks, some surprising, that prevent HDFS from reaching maximum performance. We focus on the read path of HDFS, because it can easily become a performance bottleneck for an application's input stage due to accessing comparatively slower storage media. Subsequent stages of an application are typically speed up using in-memory storage techniques and are oftentimes fast because they process less data than the input stage.

Central to our exploration is the storage access pattern of a single HDFS read request: single threaded, sequential access to large files (hundreds of megabytes) using buffered I/O. This access pattern is simple but tried and tested and has remained unchanged since the beginnings of HDFS. It is increasingly important to understand whether even a single HDFS read request using this access pattern can extract by itself the maximum throughput that the underlying storage can provide. As many big data processing systems are heavily I/O provisioned and the ratio of cores to disks reaches 1, relying on task-level parallelism to generate enough parallel requests to saturate storage is no longer sufficient. Moreover, relying on such parallelism is detrimental to application performance since each of the concurrent read requests is served slower than it would be in isolation.

We start our analysis by identifying and removing software bottlenecks present in other layers and that limit HDFS. These are compute bottlenecks introduced by the processing frameworks, network bottlenecks introduced by the OS configuration as well as performance interference caused by the local file system. We show that with simple changes HDFS reads can fully utilize the underlying storage.

Having decoupled HDFS performance from software bottlenecks we are now able to clearly identify and analyze several surprising hardware performance bottlenecks and performance variability sources induced by device internals. We focus on SSDs in this work. First, we find that the read throughput obtained from reading the same file over time can significantly and irreversibly decrease due to an increase in the per-request latency inside some SSDs. Importantly this occurs on a file-level and not for an entire device. We call this permanent slowdown. Second, we find temporary and periodic read throughput slowdown. While this temporary slowdown bears the hallmarks of SSD garbage collection we find, surprisingly, that it can occur even in the complete absence of writes. This temporary slowdown is also caused by an increase in the per-request latency inside the drive. Third, we find that file fragmentation at the file system level (ext4) introduces performance variability and slowdown during reading from every different file system extent.

All of the above findings show that the HDFS read access pattern, while generally sufficient to maximize SSD performance, can fail to do so in certain cases. Normally, HDFS maximizes SSD performance during reads because its access pattern provides enough opportunity to leverage the internal SSD parallelism. When, as discussed above, latencies inside the drive increase, the same HDFS read access pattern does not adapt by increasing the I/O request-level parallelism and this results in a throughput loss. With the help of the Flexible I/O Tester (FIO) [5] synthetic benchmark, we further analyze several OS-based approach to compensate for the throughput loss.

With experiments on 3 SSD based systems, we show that each of the slowdown we identified can individually introduce at least 10% read throughput degradation. The effect is cumulative. In the worst case, all slowdowns can overlap leading to a 30% throughput loss.

Our primary contributions are:

- We decouple HDFS read performance from several compute and configuration bottlenecks and show that it can maximize SSD read throughput.

- We identify and analyze the permanent slowdown affecting reads from the same file over time.
- We identify and analyze temporary slowdowns that affect reads even in the absence of writes.
- We identify and analyze the fragmentation slowdown affecting reads crossing to every new file system extent.

3.2 Methodology

In this section, we describe the hardware and software settings, tools, workloads, and metrics used in our analysis.

3.2.1 Removal Software Bottlenecks

We now detail the configuration changes we made to alleviate network and compute bottlenecks affecting HDFS and to eliminate sources of interference. As a result, we observed that HDFS can extract the maximum performance that our SSDs can generate.

File system configuration. We disabled access time update, directory access time update, and data-ordered journaling (we use write-back journaling) in ext4. This removes sources of interference so that we can profile HDFS in isolation.

OS configuration. Small socket buffer sizes limit the number of packets that the DataNode can send to a task and thus reduce performance by interrupting disk reads and inducing disk idleness. We increase the socket buffer size for both reads and write to match the size of the HDFS blocks.

HDFS configuration. We configured `io.file.buffer.size` to be equal to the HDFS block size. The default value of this parameter (64KB) results in too many sendFile system call, which in turn create a lot of context-switching between user and kernel space, which results in idleness for the IO device. We modified the HDFS code to allow the parameter to be set to 256MB as by default the maximum size is 32MB.

3.2.2 Experimental Setup

Hardware. We use three types of machines: *Machine A* has 2 Intel Xeon 2.4GHz E5-2630v3 processors, with 32 cores in total, 128GB of RAM, and a 450GB Intel DC S3500 Series (MLC) SATA 3.0 SSD. *Machine B* has 4 Intel Xeon 2.7GHz E5-4650 processors, with 32 cores in total, 1.5TB of RAM, and a 800GB HP 6G Enterprise SATA 3.0 SSD. *Machine C* has 2 Intel Xeon 2.4GHz E5-2630v3 processors, with 32 cores in total, 128GB of RAM, and a 512GB Samsung 860 Pro (V-NAND) SATA 3.0 SSD.

Our SSDs have been very lightly used throughout their lifetimes. After concluding our analysis we computed the total lifetime reads and writes performed on the drives using the "sectors read" and "sectors written" fields in /proc/diskstats in Linux. The value was less than 1TB for both reads and writes for each drive. This is orders of magnitude less than the manufacturer provided guarantees for SSDs. Thus, past heavy use of the drives is not a factor in our findings. Moreover, the disk utilization of our SSDs in the experiments is very low, under 20%.

Software. We use Ubuntu 16.04 with Linux kernel version 4.4.0. As a local file system we use ext4, one the most popular Linux file systems. We use HDFS version 2.7.1.

Monitoring Tools. To monitor I/O at the storage device, we rely on block layer measurements using `blktrace` and `blkparse` [3]. Blktrace collects I/O information at the block layer, while blkparse makes the traces human readable. Where necessary we use `perf` [10] and `strace` [11] to analyze program behavior.

Workloads. We use HDFS via Hadoop where we run a simple WordCount job. The exact type of Hadoop job is inconsequential for our findings because we have already decoupled HDFS performance from software bottlenecks in Section 3.2.1. We modified the Hadoop WordCount job to not write any output so that we can reliably measure read performance. We use the FIO [5] tool for analysis beyond HDFS. The data read by HDFS (or FIO) is composed of randomly generated strings and is divided in 8 ext4 files of 256MB each.

Presenting slowdowns. For every slowdown we are able to separate its effect and present results for periods with and without that slowdown. The results without a slowdown include the effects of all other slowdowns that occur at shorter timescales. For example, when comparing results with or without temporal slowdown, the results include the effect of intrinsic slowdown but not that of permanent slowdown. This is acceptable because the effect of a slowdown is roughly constant over longer periods of time.

What we measure. We evaluate performance at the HDFS DataNode level. We measure the throughput of HDFS reads and the latency of individual block layer I/O requests. As our focus is on understanding storage I/O in HDFS, we do not measure end-to-end performance related to compute frameworks' (*e.g.*, Hadoop, Spark) tasks. This allows us to avoid overheads (*e.g.*, compute, network) that compute frameworks can introduce on top of HDFS. Nonetheless, we believe that the I/O effects observed on HDFS will have an impact on I/O intensive jobs on different frameworks, no matter other overheads introduced by other resources. Before every experiment we drop all caches to ensure reads actually come from the drive.

The Hadoop tasks are collocated with the HDFS DataNodes on the same machines. The DataNodes send data to the tasks via the loopback interface using the sendfile system call. We also analyzed short-circuit reads which enable Hadoop tasks to read local input directly (using standard read calls) by completely bypassing HDFS but the findings remained the same.

Generating single requests. In our experiments using buffered I/O, two requests overlap in the device driver. In such a case, increases in request latency could be caused either by drive internals or by a sub-optimal request overlap. To distinguish such cases we performed experiments in which we tweak buffered I/O to send one request at a time. To send a single request of size X we first set the Linux read ahead size to X KB by tuning `/sys/block/<device>/queue/read_ahead_kb`. We then use the `dd` command to read one chunk of size X (dd bs=X count=1). The influence of read ahead size on block layer size is known and discussed in related work [54].

3.2.3 Metrics

For the rest of this section, the word "file" refers to one 256MB ext4 file. In HDFS parlance this represents one HDFS block.

File throughput. The number of bytes, read from the target file, divided by the period of time. The period starts with the submission of the first block layer I/O request in the file (as timestamped by blktrace) and finishes with the completion of the last block layer I/O request in the file (as timestamped by blktrace). During this period we only count time when the disk is active, i.e. there is at least one I/O request being serviced by or queued for the drive. This metric removes the impact of disk idle time caused by context-switches between user and kernel space in the application. Our HDFS results show no disk idle time after applying the changes in Section 3.2.1 Nevertheless, disk idle time appears in FIO and we chose to discard it for a fair comparison to HDFS. Overall, the disk idle time does not influence our main findings.

Request Latency. The time between the timestamp when a block layer request is sent to the drive (D symbol in blktrace) and the timestamp of its completion (C symbol in blktrace). Both timestamps are taken from blktrace.

Fragmentation. The number of extents an ext4 file has. Note that all of our files are 256MB (frequently used by industrial workloads [76, 77, 115]). The maximum extent size in ext4 is 128MB. Thus, the minimum possible number of extents in a file is 2. This file size, in conjunction with ext4 extent size, allows us to observe the behavior of smaller files sizes (*i.e.*, each extent of 128MB). Note that we do not study smaller files sizes (*e.g.*, <1MB) as HDFS was not designed for these workloads.

Recovery Time. The period of time during which I/O requests have higher than usual latency due to intrinsic slowdown. This is measured starting from the first I/O read request of an ext4 extent until either the latency of the requests decreases to normal or the extent is fully read, whichever comes first.

3.2.4 Maximizing Device Throughput

An important goal for HDFS is to maximize the throughput obtained from the storage devices. One way to achieve this is via multi-threading in the DataNode. This is already part of the design as different DataNode threads can serve different task requests concurrently. While this can maximize device throughput it does so at the expense of single-thread performance which reduces task performance.

State-of-the-art data processing systems are heavily I/O provisioned, with a ratio of CPU to disk close 1 [9]. In this context, relying on parallelism to make the most of the storage is unlikely to help because the number of tasks is roughly the same as the number of disks (tasks are usually scheduled on a separate core). As a result, it is important to understand and ensure that the HDFS access pattern (single-thread, large files, sequential access, buffered I/O) can by itself extract maximum performance from SSDs.

3.3 Intrinsic Slowdown

In this section, we introduce the intrinsic slowdown, a performance degradation that predictably affects files at short time scales (seconds to minutes). This slowdown is related to the logical file fragmentation. Every time a new file system extent is read, a number of I/O requests from the start of the extent are served with increased latency and that is correlated with throughput drops. Interestingly, even non-fragmented files are affected since a file has to have at least one extent and every extent is affected by the slowdown. The more fragmented a file is, the more extents it has, and the bigger is the impact of the slowdown.

Intrinsic slowdown appears on all the machines we tested and causes a drop in throughput of 10-15% depending on the machine. The slowdown lasts a variable amount of time but there is no correlation with extent size. This slowdown affects not only HDFS but all applications using buffered I/O.

The remainder of this section presents an overview of the throughput loss, an analysis of the results, a discussion on the causes, and an analysis of the mitigation strategies.

3.3.1 Performance Degradation at Glance

Figure 3.1 illustrates the influence that an increased number of extents has on throughput for each of the 3 machines. In this figure, each point represents the average file throughput of a set of files with the same number of extents in one machine. Files were created using ext4's default file allocation policies so we had no control over the number of extent each file was allocated. We observed that ext4 allocations result in highly variable fragmentation levels even on our drives which were less then 20% full. We often saw cases where one file was allocated 30 extents and a file created seconds after was allocated 2 extents. A thorough analysis of ext4 allocation patterns is, however, beyond the scope of this work.

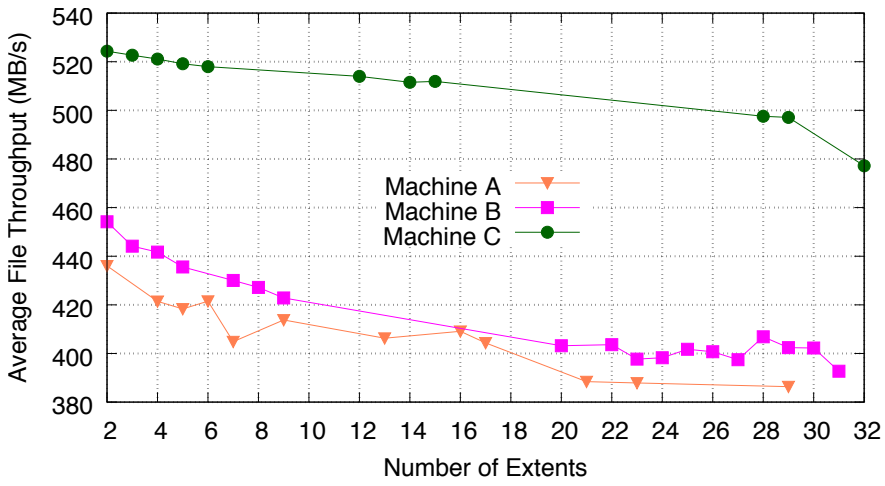


Fig. 3.1 HDFS read. Average file throughput vs number of extents.

The figure shows that an increase in fragmentation is correlated with a loss in throughput. This finding holds on all 3 machines but the magnitude of the throughput loss is different because the SSDs are different. With 29 extents, throughput drops by roughly 13% for machines A and B, but by less than 5% for machine C. There is a limit to the throughput loss and that is best exemplified by the fact that throughput drops very slowly for machines A and B after 20 extents. The reason is that the extents are smaller but the recovery period is not correlated with the extent size so a very large percentage of the I/O requests is affected by the slowdown.

Correlations. We next analyze I/O request latency. Figure 3.2 presents the request latencies on machines A, B and C, during an HDFS read. The dashed lines correspond to request latencies after the intrinsic slowdown disappeared while the continuous one show latencies during the slowdown. Latencies increase during slowdown both at the median but especially in the tail. Machine C shows both the smallest latencies and the smallest degradation and this is due to the fact that its SSD is based on a different technology (V-NAND).

The above latencies correspond to the standard buffered I/O configuration in which 2 requests overlap in the device driver. We also measured (not illustrated) the request latency when

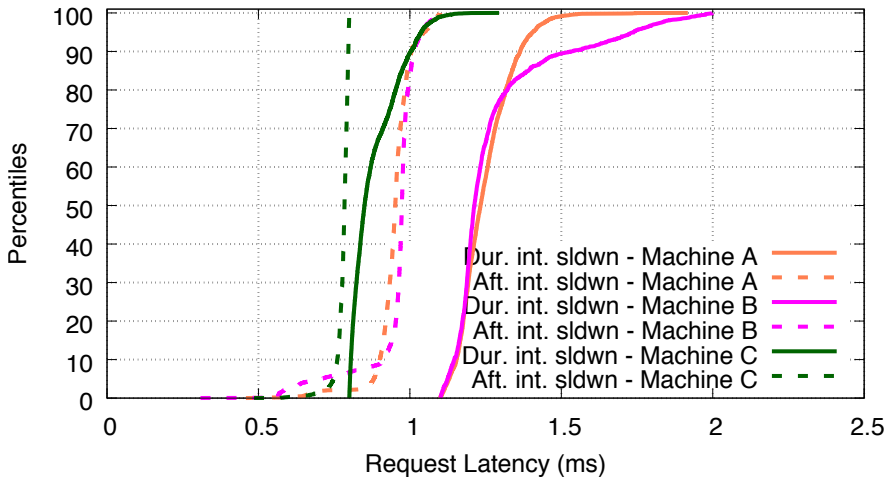


Fig. 3.2 HDFS read. Request Latency CDFs during and after intrinsic slowdown.

sending one single request a time. We find that latency remains unaffected even during the parts of the extent that are normally affected by intrinsic slowdown. This suggests that the latency increase and throughput loss are not solely due to drive internals. This is expected as the SSD FTL has no notion of extents which are a file system construct.

We find that the inherent slowdown is correlated with a sub-optimal request overlap in the device. Consider a request R_2 and let S_2 and E_2 be its en-queueing and completion time. With buffered I/O, the execution of R_2 overlaps with the final part of R_1 and the first part of R_3, R_1 being the previous request and R_3 the next. We have that $S_2 < E_1 < S_3 < E_2$. We find that periods of intrinsic slowdown are correlated with an imbalanced overlap, that is R_2 overlaps much more with either R_1 or R_3. In other words, imbalance overlap occurs when $T >> 0$ where $T = abs((E_2 - S_3) - (E_1 - S_2))$. To exemplify, Figure 3.3 shows the correlation between T and request latency for a sample extent. For the first 20 requests, the overlap is sub-optimal and latency suffers. The overlap imbalance is corrected around request 22 and soon after latency drops under 1ms which is the latency we normally see outside of intrinsic slowdown. The following paragraphs describe the intuitions behind the overlap imbalance.

Fig. 3.3 Correlation between increased latency and request overlap imbalance.

Characterization of recovery periods. We next analyze the duration and variability of the recovery periods. There are two main insights. First, even for a single extent size and one machine, there can be significant variation in the duration of the recovery period. Second, the duration of the recovery period is not correlated to the extent size.

Figure 3.4 shows CDFs of the duration of the recovery period on the 3 machines. For each machine we show large extents (128MB) with continuous lines and smaller extents (32-40MB) with dashed lines. We aggregated results from extents from multiple files if they have the target size and reside on the same machine. We measure the recovery duration in number of requests. The request size is 256 KB.

The CDFs for any one of the machines show a similar pattern in the recovery period despite the different extent size. Therefore, extent size is not a factor with respect to the duration of the recovery period.

There is significant variability in the recovery period for every extent size on machines A and B. The worst case recovery duration is more than 5$\times$ that of the best case. In contrast, machine C shows much less variability.

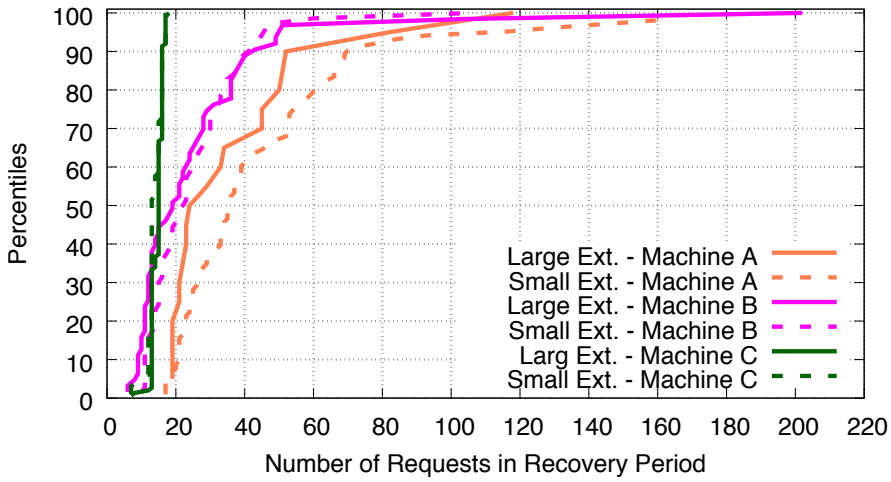


Fig. 3.4 HDFS read. CDFs of number of requests executed in the recovery period for large and small extents.

If we compute the recovery period relative to extent size (not illustrated) we find that for the smallest extents (*e.g.*, 8MB) it is common for at least 50% of the requests in the extent to be affected by intrinsic slowdown. In the worst case, we have seen 90% of an extent being affected.

Discussion on internal SSD root cause. Since we do not have access to the proprietary SSD FTL design we cannot directly search for the root cause internal to the drive. We believe that sub-optimal request overlap leads to throughput loss because it forces the drive to be inefficient by serving both overlapping requests in parallel when the most efficient strategy would sometimes be to focus on the oldest one first. The request stream enters in this state due to the initial requests at the start of the extent. The stream self-corrects by eventually reaching the optimal (balanced) request overlap and remaining there, in our setting, an optimal I/O request latency corresponds to $<$ 1ms. The software does not help in the correction as it functions in a reactive manner. It sends a new request as soon as one completes. The self-correction happens solely due to timing, based on the request latencies. This also explain the variability in the recovery periods.

Discussion on OS root cause. Operating systems provide abstractions (*e.g.*, system calls) to improve the performance and efficiency of resources under particular workloads. In particular, the Linux kernel attempts to improve the I/O performance of sequential workloads via the readahead functionality [103]. In Linux, the readahead size is adaptive [1], thus the requests generated by this technique can vary in size and behavior, depending on the system state (*e.g.*, memory). In further paragraphs, we show that configuration parameters related to readahead (*i.e.*, request size) can mask the intrinsic slowdown, mainly by increasing request parallelism. Analyzing behavior of the readahead implementations is out of the scope of this thesis, and it is left for future work.

3.3.2 Mitigation Strategies

We consider mitigation strategies that are more aggressive in generating request level parallelism in the hope that they could compensate for the loss in throughput due to the slowdown. We find that both direct I/O as well as increasing the number of requests sent in parallel with buffered I/O can mask intrinsic slowdown. Our evaluation indicates that an increase in request parallelism can amortize the request processing latency delay. Hence, the root cause of the initial tail latency seems to be due to the relationship between OS behavior (*i.e.*, buffered I/O) and drive behavior (*i.e.*, error correction and request processing).

Figure 3.5 compares average file throughput vs number of extents, when using direct I/O across different machines. The files are the same as in Figure 3.1. The figure shows that average throughput is maintained across different numbers of extents with direct I/O. The tendency holds across all machines tested. In other words, direct I/O can mask intrinsic slowdown. The reason is that by sending more and larger requests, direct I/O better leverages the device parallelism. We observe the same effect when increasing parallelism in buffered I/O, by increasing the read ahead size. This setting results in both larger requests as well as more requests being sent in parallel to the drive.

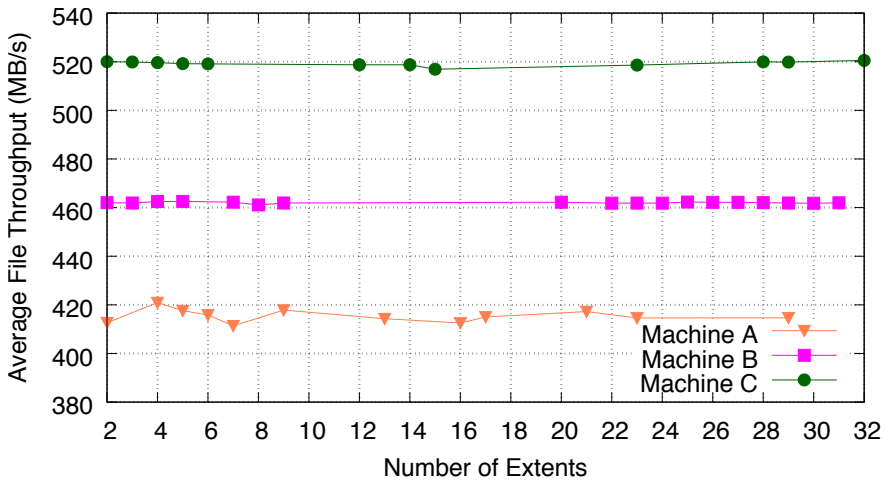


Fig. 3.5 FIO Direct I/O read. Average file throughput vs number of extents.

3.4 Temporal Slowdown

In this section, we introduce the temporal slowdown, a periodic and temporary performance degradation that affects files at medium timescales (minutes to hours). At the high level, the pattern in which temporal slowdown manifests might we confused with write-induced SSD garbage collection (GC). However, temporal slowdown is not always GC. Surprisingly, on machine A, it always manifests even in read-only workloads. On machine B, it is indeed triggered by writes but interestingly it takes a very small amount of writes relative to the drive capacity to trigger temporal slowdown. Moreover, our SSDs have a very low utilization (under 20%). We link the slowdown to tail latency increases inside the drive. Temporal slowdown causes a throughput drop of up to 14%. Temporal slowdown affects not only HDFS, but all applications using either direct or buffered I/O.

The remainder of this section presents an overview of a throughput loss, an analysis of the results, a discussion on causes, and an analysis of mitigation strategies.

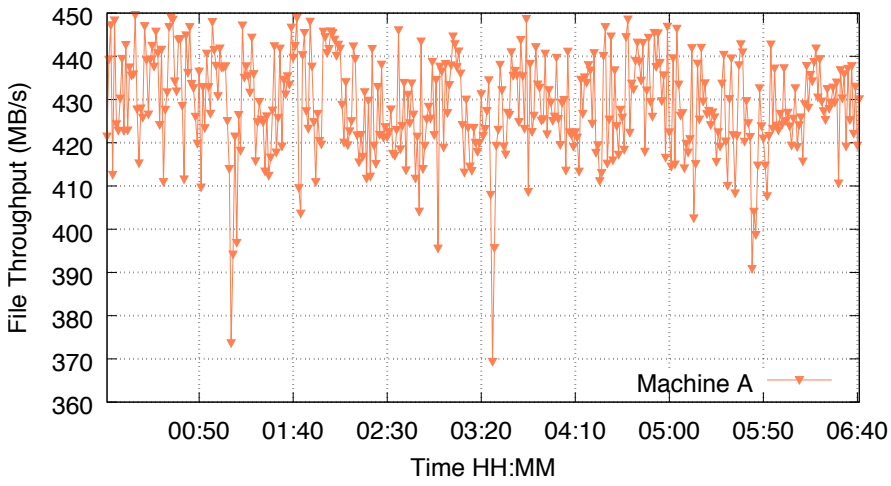


Fig. 3.6 HDFS read. File throughput timeline on Machine A.

3.4.1 Performance Degradation at a Glance

Figure 3.6 presents the throughput timeline of a file affected by temporal slowdown on machine A. It shows three instances of the slowdown around the 1:00, 3:30 and 5:40 marks. The rest of the throughput variation is caused by inherent slowdown. The average throughput of the periods not affected by the slowdown is 430MB/s. The first instance of slowdown causes a drop in throughput to 370MB/s, a 14% drop from the 430 MB/s average. On machine A, temporal slowdown appears on average every 130 min and last on average 5 min.

Figure 3.7 shows the same experiment on machine B. There are 5 instances of temporal slowdown clearly visible due to the pronounced drops in throughput. The average throughput of the periods not affected by the slowdown is 455MB/s. The biggest impact is caused by the third slowdown instance which causes a drop to 390MB/s, almost 15% down from the average. On machine B, temporal slowdown appears on average every 18 minutes and last for 1.5 minutes.

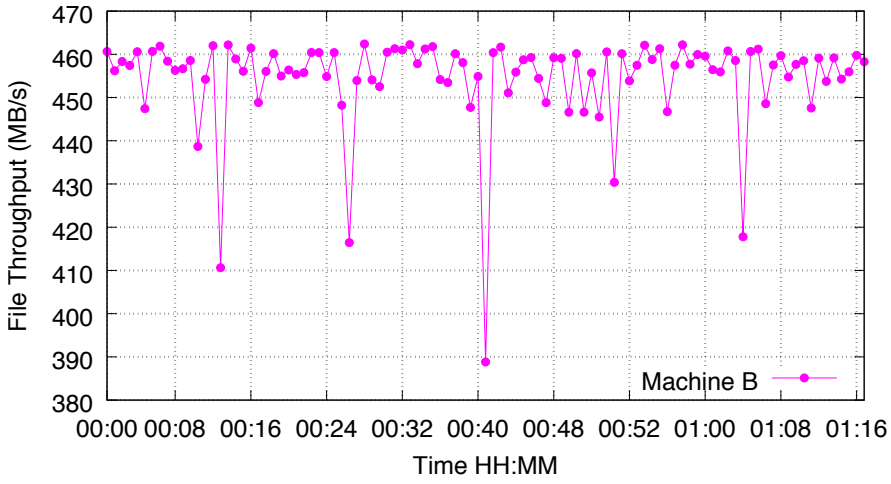


Fig. 3.7 HDFS read. File throughput timeline on Machine B.

3.4.2 Analysis

Correlations. We next analyze I/O request latency. Figure 3.8 shows a CDF of the request latencies for one file. One line shows latencies during temporal slowdown while another shows latencies during periods not affected by the slowdown. The difference lies in the tail behavior. During temporal slowdown a small percentage of the requests show much larger latency. This is consistent with the impact of background activities internal to the drive.

The experiments in Figures 3.6 and 3.7 introduce writes and they responsibility on triggering temporal slowdown on machine B. Even though from the application perspective (*i.e.*, Hadoop) the workload is read-only, a small number of writes appear due to HDFS metadata management. These are the only writes in the system as we explicitly turned off journaling and metadata updates in ext4. Interestingly, a small amount of writes relative to the drive size is sufficient to trigger temporal slowdown. On machine B, temporal slowdown occurs approximately every 120MB. That amounts to only 0.015% of the disk size.

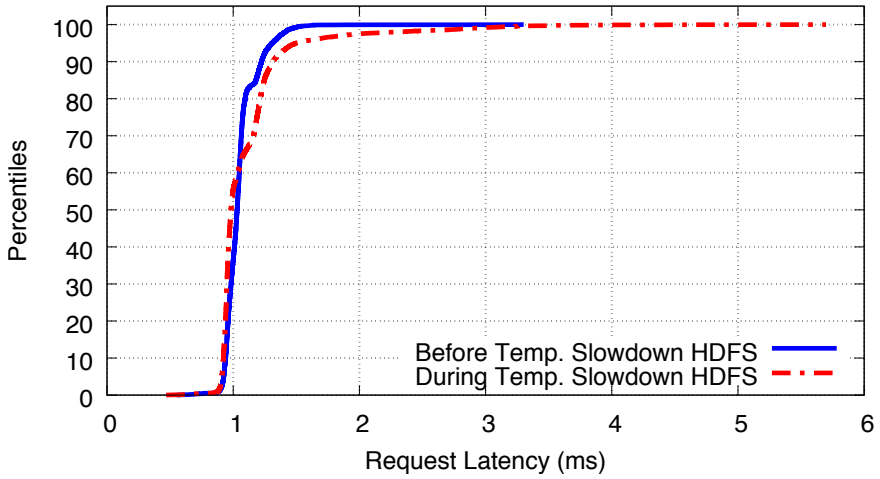


Fig. 3.8 HDFS read. CDFs of read request latencies during and outside of temporal slowdown on Machine A.

Temporal slowdown without writes. Our main finding related to temporal slowdown is that it can occur in the complete absence of writes. This occurs only on machine A so we focus on it for these experiments. To avoid any writes, we repeat the experiment using FIO instead of HDFS. We configure FIO to use the `read` system call and evaluated both direct I/O and buffered I/O. The results were similar so we only show direct I/O. We confirm that there are no writes performed during the experiments by checking the number of written sectors on the drive (from `/proc/diskstats`), before and after the experiments. In addition, we ensure that no writes have been performed in the system for at least one hour before the start of the experiments.

In Figure 3.9, we show the throughput timeline when using FIO with direct I/O. FIO shows more variability in the common case compared to Hadoop because of context switches between kernel and user space. The temporal slowdown is again visible despite the absence of writes. The slowdown appears every 130 minutes on average and lasts 5 minutes on average. The periodicity is almost identical to the HDFS case, suggesting that the HDFS metadata writes did not play a role in triggering temporal slowdown on machine A.

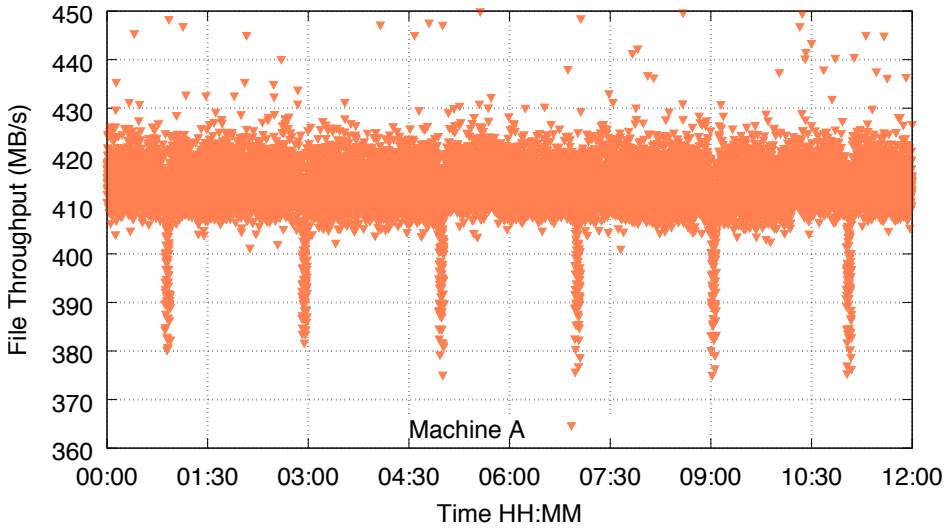


Fig. 3.9 FIO direct I/O Read. File throughput timeline on Machine A.

Figure 3.10 presents the I/O request latency for FIO with direct I/O. Again, tail latency increases during slowdown. The four different latency steps appear because direct I/O sends, by default, four large block layer requests (1MB) to the drive. Note that we also performed experiments to discard the I/O scheduler as the root cause for the latency delays. Different I/O schedulers (*e.g.*, deadline, NOOP, CFQ) were evaluated. The experiments did not reveal any significant difference in the latency distributions between distinct I/O schedulers. Hence, we present the I/O request latencies for the deadline scheduler (the default I/O scheduler in linux).

Trigger of slowdown without writes. Next, we analyze whether temporal slowdown in the absence of writes is correlated with the number of reads performed or it is time-based. We introduce periods of inactivity using sleep periods between the reads. We make sure that these periods are much smaller than the duration of temporal slowdown so that we do not miss slowdown events. We find that regardless of the inactivity period induced, the periodicity remains the same suggesting time-based triggers.

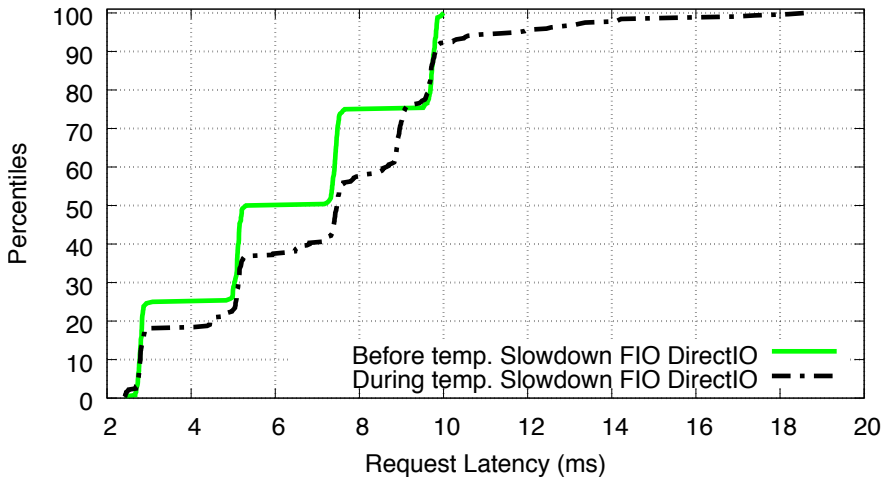


Fig. 3.10 FIO Direct I/O Read. CDFs of read request latencies before and during temporal slowdown on Machine A.

Discussion on internal SSD root cause. Since we do not have access to the proprietary SSD FTL design we cannot directly search for the root cause internal to the drive. In theory, there are three known culprits for temporal slowdowns in SSDs, yet our findings does not match any of them. The first one is write-induced GC [90, 108]. However, we show that temporal slowdown can appear in the absence of writes as well. The last two culprits are read disturbance and retention errors [31]. In the related work, in Section 3.6, we argue at length that these culprits appear on drives that are far more worn out (orders of magnitude more P/E cycles) than ours and after order of magnitude more reads have been performed. We hypothesize that temporal slowdown on our drives is triggered by periodic internal bookkeeping tasks unrelated to past drive usage or current workload.

3.4.3 Mitigation Strategies

We have found no simple way of masking temporal slowdown. It occurs for both buffered I/O and direct I/O. One could attempt to detect early signs of slowdown or estimate its start

via profiling and then avoid performing reads during the period. This would yield more predictable performance at the expense of delays.

3.5 Permanent Slowdown

In this section, we introduce the permanent slowdown, an irreversible performance degradation that affects files at long timescales (days to weeks). Permanent slowdown occurs at a *file level*. It is not triggered by a single drive-wide event. Thus, at any point in time, a drive can contain both files affected by permanent slowdown and files unaffected by it. The exact amount of time it takes for a file to be affected by permanent slowdown varies from file to file and is not influenced by how many times a file was read. We only see permanent slowdown on machines of type A. Permanent slowdown causes a throughput drop of up to 15%.

We find that permanent slowdown is not specific to HDFS, but affects all read system call that use buffered I/O. We link the slowdown to unexpected and permanent latency increases inside the drive for all I/O requests.

For terminology, in the context of permanent slowdown, "before" means before the first signs of slowdown and "after" means after slowdown completely set in. The CDFs represent a single HDFS file composed of 8 blocks (*i.e.*, 8 ext4 files). Figure 3.11 shows a different file where we caught the onset of the slowdown. Nevertheless, we have seen that all files affected by the slowdown show a similar degradation pattern and magnitude.

The remainder of this section presents an overview of a throughput loss, an analysis of the results, a discussion on causes and an analysis of mitigation strategies.

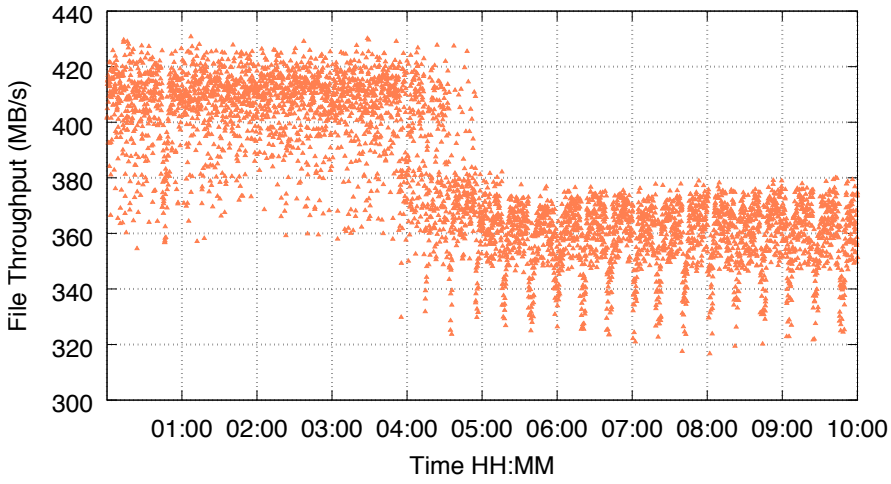


Fig. 3.11 HDFS read. File throughput over a 10 hour period centered around the onset of permanent slowdown.

3.5.1 Performance Degradation at a Glance

Figure 3.11 shows the onset and impact of permanent slowdown. The plot shows a 10 hour interval centered around the onset of permanent slowdown. The file was created several days before this experiment was ran. For the first 4 hours, read throughput lies between 340 MB/s and 430 MB/s. This variation is explained by the intrinsic and the temporal slowdowns described in Sections 3.3 and 3.4. Around the fourth hour, the permanent slowdown appears and after less than one hour it completely sets in. From that point on, read throughput remains between 320 MB/s and 380 MB/s in this experiment and all future experiments involving this file.

Figure 3.12 compares the CDF of the read throughput of the same file before and after slowdown. At the median, throughput drops by 14.7% from 418 MB/s to 365 MB/s.

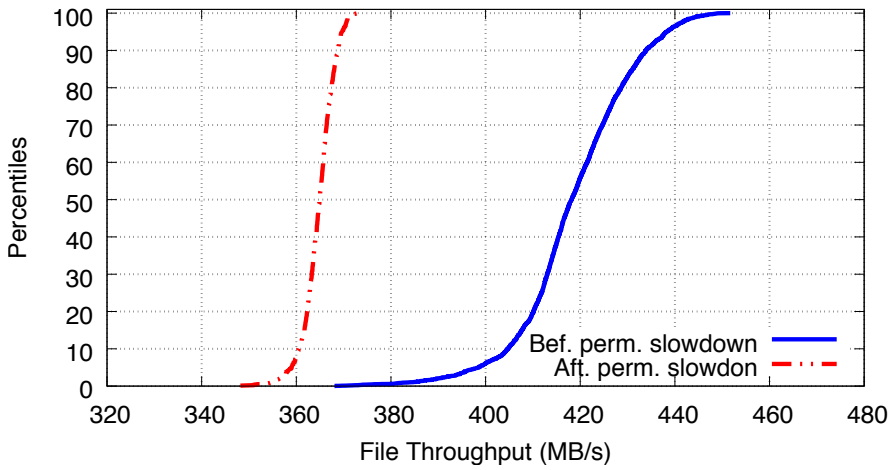


Fig. 3.12 HDFS read. CDFs of file throughput before and after permanent slowdown.

3.5.2 Analysis

Generality. We start by analyzing the generality of the permanent slowdown. HDFS uses the `sendfile()` system call to transfer data. Using the `perf` tool we find that `sendfile()` shares most of its I/O path in the Linux kernel with the read system calls that use buffered I/O. Therefore, we ask whether permanent slowdown affects only `sendfile()` system calls or also read system calls that use buffered I/O.

We use FIO to generate reads using buffered I/O. We configure FIO to use the read system call (*i.e.*, sync io engine as a FIO parameter). Figure 3.13 presents a throughput comparison between HDFS (the `sendfile()` system call) and FIO (read system call). The two rightmost CDFs show the throughput for HDFS and FIO before permanent slowdown. HDFS and FIO behave similarly. The same applies after permanent slowdown sets in (leftmost CDFs). Similar results were obtained using libaio as an I/O engine for FIO. This result show that permanent slowdown does not affect a particular system call (`sendfile()`) but the group of read system calls that perform buffered I/O.

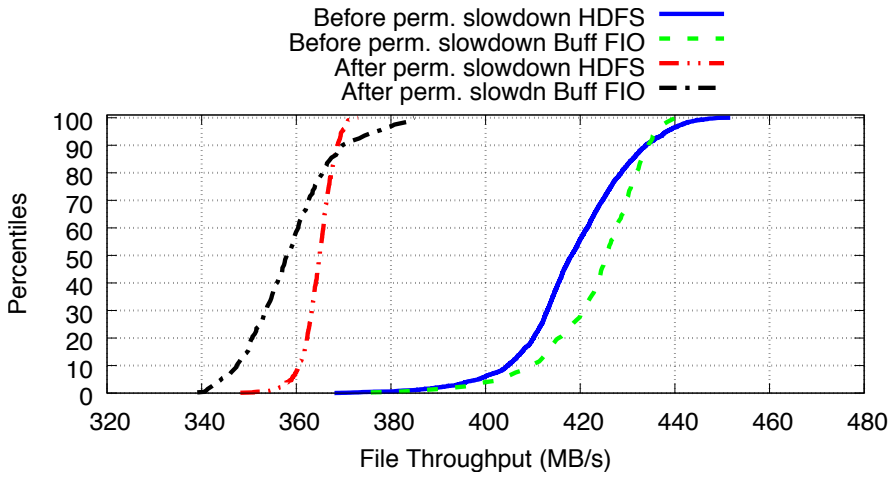


Fig. 3.13 HDFS read vs Buffered FIO. CDFs of file throughput before and after permanent slowdown.

Correlations. We next analyze I/O request latency. Figure 3.14 compares the CDFs of request latency in HDFS on one file before and after permanent slowdown. Permanent slowdown induced an increase in latency at almost every percentile. Thus, most requests are treated slower. At the median, the latency increases by 25%. The latencies in the tail of the CDF are explained by the inherent and the temporary slowdowns. We re-run the experiment using FIO and saw similar results.

We also measure latency when sending one request at a time. We vary request size between 128KB and 1MB. We find that single request latency also increases after permanent slowdown and for all sizes. The increase in latency is constant in absolute terms and is thus not correlated to request size. The latency for the default request size of 256KB increased by 33%. These findings show that latency increases are due to the drive and not due to the software layers.

Discussion on internal SSD root cause. The read disturbance and retention errors discussed as potential culprits for temporary slowdown could conceivably lead to permanent slowdown [31] if left uncorrected by the drive. However, the same argument we made for temporal slowdown applies. Read disturbance and retention occur on drives much more

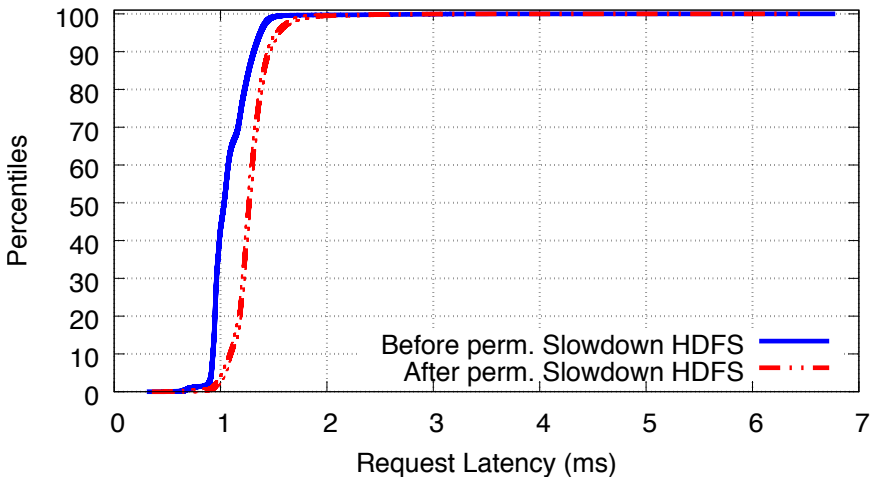


Fig. 3.14 HDFS read. CDFs of request latency before and after permanent slowdown.

worn out (orders of magnitude more P/E cycles) than ours and after performing orders of magnitude more reads. We hypothesize that the reason for permanent slowdown lies with error correction algorithms being triggered inside the drive after enough time has passed since file creation. Note that once the permanent slowdown presents itself, file access is consistently slower. This behavior slowdown persists across time, even when file access stops (*e.g.*, for a period of days, weeks, or months).

Discussion on internal OS root cause. The increase of request parallelism via operating system configuration parameters. More specifically, tunning the configuration values that are pertitnent to the readahead optimization [80] (*i.e.*, request size), similar to the intrinsic slowdown, masks the performance degradation caused by the permanent slowdown. However, unlike the previous slowdowns, permanent slowdown is not present from the beginning of file lifetime. Therefore, oeprating system configuration values, in particular local file system behavior cannot be signaled as the sole cause of the slowdown. Instead, we believe that particular OS behaviors in conjuction with internal drive behaviors cause the permanent slowdown.

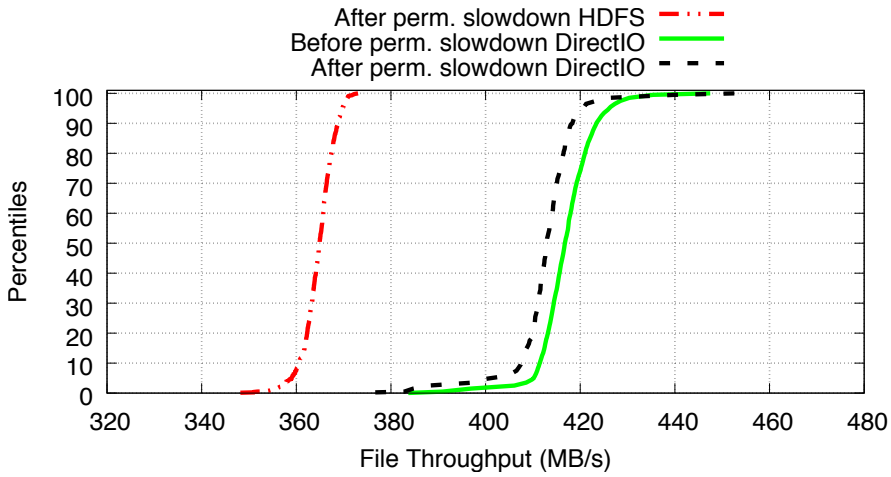


Fig. 3.15 HDFS read vs FIO Direct I/O. CDFs of file throughput before and after permanent slowdown.

3.5.3 Mitigation Strategies

We consider mitigation strategies that are more aggressive in generating request level parallelism in the hope that they could compensate for the throughput loss. We find that both direct I/O as well as increasing the number of requests sent in parallel with buffered I/O can mask permanent slowdown.

First, we look at the behavior of permanent slowdown when reading with direct I/O. It is known that direct I/O issues more and larger requests to the block layer, when compared to buffered I/O [54]. In our experiments it issues four 1 MB in parallel. Figure 3.15 presents a throughput comparison between HDFS and FIO with direct I/O. The two rightmost CDFs correspond to the throughput of FIO with direct I/O before and after permanent slowdown. The difference between the two is minimal. We repeated the experiment using a smaller, 256KB direct I/O request size (by tuning `/sys/block/<device>/queue/max_sectors_kb`). The results remained the same suggesting that having a larger number of parallel requests is key for best performance.

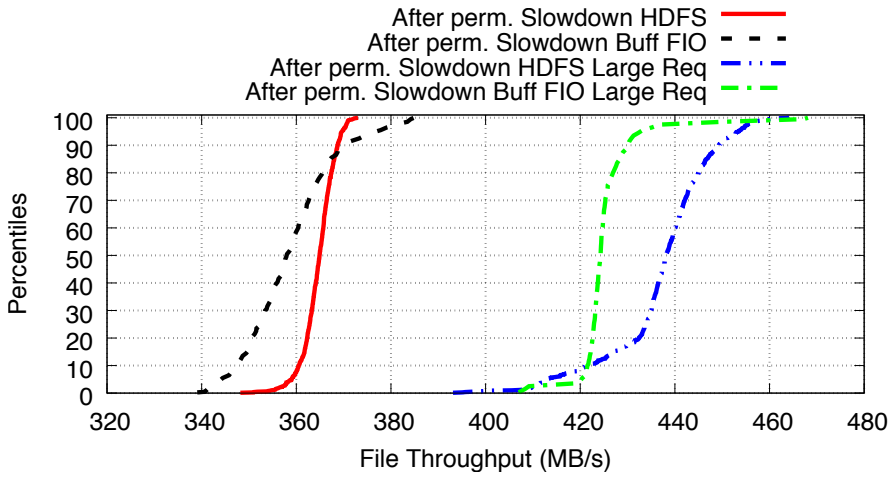


Fig. 3.16 HDFS read vs FIO Buffered I/O. CDFs of file througput. Before permanent slowdown with I/O requests of 256KB, and after permanent slowdown with large I/O requests of 2MB.

We also analyzed making buffered I/O more aggressive. We increase the request size from 256KB to 2MB by modifying the read ahead value. This change automatically brings about a change in the number of request sent in parallel to the drive. When request size is 256KB, two requests execute in parallel. For a 2MB request, four parallel execute in parallel. Figure 3.16 presents four CDFs representing the throughput after permanent slowdown with HDFS reads and FIO buffered reads. The leftmost CDFs correspond to the default request size of 256KB and show the impact of permanent slowdown. The rightmost CDFs are for a request size of 2MB. The modified buffered I/O is able to mask the permanent slowdown with increased parallelism.

3.6 Related Work

Related to sources of performance variation internal to SSDs. Garbage collection (GC) in SSDs is known to trigger temporary slowdowns but it is write induced [90, 108]. Flash on

Rails [90] reports no GC-like effects in read-only workloads. Since our paper focus solely on read-only workloads we do not discuss further GC.

There are two types of errors that can appear in read-only workloads, retention errors and read errors. Retention errors occur when data stored in a cell changes as time passes and are caused by the charge in a cell dissipating over time through the leakage current [31, 33]. Read (disturbance) errors occur when the data in a cell is modified over time as a neighboring cell is read repeatedly and are caused by the repeated reads shifting the threshold voltages of unread cells and switching them to a different logical state [31, 32]. In practice, retention errors happen much more frequently than read disturbance errors [75].

The temporary slowdowns we encountered show a different pattern compared to the two read errors described above. Related work shows that read errors are highly correlated with the number of P/E cycles that the drive went through [31]. Our drives have a very low P/E cycle. At the end of our experiments, the amount of data written to the drives over their entire lifetime was just 1 TB, double their capacity. In contrast, related work uses drives with thousands of P/E cycles to show a noticeable increase in error rates [31]. Similarly, to obtain read errors, related work [32] perform hundreds of thousands of reads on a single page in order to see noticeable effects. Our experiments perform at most a few thousand reads. In addition, the read-errors results from related work [32] are on drives that already underwent thousands of P/E cycles.

Gunawi *et al.* [50] study 101 reports of fail-slow hardware (some of which SSD-related) incidents, collected from large-scale cluster deployments. One the SSD front, they find firmware bugs that cause latency spikes or stalls and slow reads due to read retries or parity-based read reconstruction. The study finds that slow reads occurs mostly on worn out SSDs or SSDs that approach end of life. We show that similar problems can occur on very lightly used SSDs. Moreover, we analyze the impact that these hardware issues have at the application level.

Jung *et al.* [61] find at least 5× increased latency on reads when enabling reliability management on reads (RMR). RMR refers collectively to handling read disturbance management, runtime bad block management, and ECC. The latency differences causing the slowdown we uncover are much less pronounced. Moreover, our slowdowns illustrate dynamics in read latency over time whereas this work focuses on read-related insights from parameter sweeps.

Hao *et al.* [52] perform a large-scale study analysis of tail latency in production HDDs and SSDs. Their study presents a series of slowdowns and shows that drive internal characteristics are most likely responsible form them. They characterize long slowdown periods that (may) last hours and affect the whole drive, without any particular correlation to I/O rate. Like them, we find that the drive is most likely responsible for most of the slowdowns. We did not experience large period of slowdowns across the whole drive, probably due to the fact that our drives are more lightly used.

Related to fragmentation in SSDs. Conway *et al.* [40] show that certain workloads cause file systems to age (become fragmented) and this causes performance loss even on SSDs. Their workloads involve many small files (<1MB). Kadekodi *et al.* [63] also exposes the impact of aging in SSD across a variation of workloads and file sizes. They focus on replicating fragmentation to improve benchmarking quality. Similarly, Chopper [55] studies tail latencies introduced by block allocation in ext4 in files of maximum 256KB. In contrast, we study intrinsic slowdown in much larger files (256MB) and we quantify the impact on HDFS.

Related to extracting best performance out of SSDs. He *et al.* [54] focus on five unwritten rules that applications should abide by to get the most performance out of the SSDs and analyze how a number of popular applications abide by those rules. These rules boil down to specific ways of creating and writing files: write aligned, group writes by death time, create data with similar lifetimes, etc. These findings are all complementary to our work. The authors also point to small I/O request sizes and argue that they are unlikely to use the SSD parallelism well. In contrast we see that in the common case, the default I/O request sizes

can extract the maximum SSD performance but fall short when hardware behavior changes as exemplified by the permanent slowdown.

Related to storage-influenced HDFS performance. Shafer *et al.* [87] analyze the performance of HDFS v1 on HDDs using Hadoop jobs. They show three main findings. First, architectural bottlenecks exist in Hadoop that result in inefficient HDFS usage. Second, portability limitations prevent Java from exploiting features of the native platform. Third, HDFS makes assumptions about how native platforms manage storage resources even though these vary widely in design and behavior. Our findings complement this past work by looking at SSDs instead of HDDs. Moreover, we look at the influence that internal drive characteristics have on HDFS performance while this past work focuses on software level interactions. Harter *et al.* [53] study HDFS behavior under HBase workload constraints: store small files (<15MB) and random I/O. In contrast, we study HDFS under regular conditions, with large files and sequential I/O.

Related to performance variability of storage stacks. Cao *et al.* [36] study the performance variation of modern storage stacks, on both SSDs and HDDs. For the workloads they analized they find SDD performance in ext4 to be stable even across different configurations, with less than 5% relative range. In contrast we show variations of up to 30% over time, for one single configuration for HDFS. Maricq *et al.* [72] conduct a large-scale variability study. Storage wise, they focus on understanding performance variability between HDDs and SSDs. Similar to us, they find that sending large number of requests to the SSDs reduces performance variability. However, they focus on workloads with direct I/O and small request sizes (4 KB). In contrast, we study SSD variability both under direct I/O and buffered I/O. We dive deeper into the importance on the number and size of requests. Vangoor *et al.* [99] analyze the performance overheads of FUSE versus native ext4. Their analysis shows that in some cases FUSE overhead is negligible, while in some others it can heavily degrade performance. HDFS is also a user space file system, however it has a different architecture and functionality, and use cases than FUSE. In this work, we analyze the interaction between

HDFS and lower layers of the storage stack, under HDFS main use case, sequential I/O in large files.

3.7 Discussion

We showed that intrinsic and permanent slowdowns occur because software cannot adapt its strategy for extracting the maximum performance and parallelism from the device in the face of changes in SSD behavior. In the common case, software can extract the maximum performance and parallelism from the SSD using a set strategy of generating I/O requests. When SSD performance drops due to internal causes, the same set strategy cannot continue to extract the same performance level. A more aggressive strategy is needed. Yet, HDFS cannot adapt. This points to the need to consider more adaptable software designs that readjust according to perceived performance drops and instabilities in hardware.

The more aggressive approaches that we evaluated were switching to direct I/O and increasing the size and number of parallel I/O requests in buffered I/O. Unfortunately, for existing applications, especially those with large code bases like HDFS, this more aggressive approach may not always be easy to leverage. Switching to direct I/O may require extensive changes to application code. Increasing the aggressiveness of buffered I/O may lead to wasted disk reads if turned on at machine level. If turned on per application, aggressive buffered I/O may influence fairness in the co-existence with collocated workloads. In addition, from an operational perspective, increasing the aggressiveness of buffered I/O is not straight-forward. First, it is not intuitive because under the common case the default strategy for buffered I/O is enough to extract maximum performance from SSDs. Moreover, in Linux, the settings required to increase aggressiveness are controlled by a seemingly unrelated configuration that controls read ahead size.

Our findings have an impact on they way systems are benchmarked on SSDs. If two systems are tested on copies of the exact same file, 10-20% of the performance difference may come

from intrinsic slowdown (a copy is more fragmented) and/or permanent slowdown (a copy is older). Even if the same input file is used but at different points in time, 10% of the performance difference may come from permanent slowdown. Finally, if systems are tested for short periods of time, 10% of the difference can come from temporary slowdown if one of the systems is unlucky to run during one slowdown episode. In the extreme case, one system may be affected by all three slowdowns at the same time while another may only be slightly affected by intrinsic slowdown. In this case, almost 30% of the performance difference may come from the slowdowns and not the systems under test.

3.8 Summary

In this chapter we introduced and analyzed three surprising performance problems (inherent, temporal and permanent slowdowns) that stop large sequential read workloads from extracting maximum performance from some SSDs. These problems are introduced by the layers sitting beneath user-space storage systems (*e.g.*, file system, SSDs). The lower layers also hold the key to masking two of the three problems by increasing I/O request parallelism during the problems. Unfortunately, user-space storage systems do not have the ability to adapt. A large sequential read access pattern access pattern successfully extracts maximum performance from SSDs in the common case but it is not aggressive enough to mask the performance problems we found. Our results point to a need for adaptability in storage stacks.

Chapter 4

Reducing Read Tail Latency in Distributed File Systems

4.1 Motivation

Large-scale distributed systems exhibit unpredictable high-percentile (tail) latency variations, which harm performance predictability and may drive users away. Many factors may cause these variations, such as component failures, replication overhead, load imbalance, and resource contention [41, 44, 62, 69, 100]. The problem is exacerbated when systems run on harvested resources. Resource-harvesting datacenters co-locate latency-sensitive services (*e.g.*, search engines) with batch jobs (*e.g.*, data analytics, machine learning) to improve resource utilization [48, 101, 112, 115]. In these datacenters, performance isolation mechanisms [43, 48, 57, 70, 73, 101, 109, 111, 112, 115] throttle or even deny resources to the batch jobs when the services need them.

Many papers have addressed tail latency management in distributed systems. For example, they have tackled straggler tasks in data analytics frameworks [22, 23, 92, 113] and requests in multi-tier services [58], tail latencies in distributed file/storage systems [68, 93], and enforced service-level objectives (SLOs) for compute or storage [62, 71, 97, 102, 104, 105, 116].

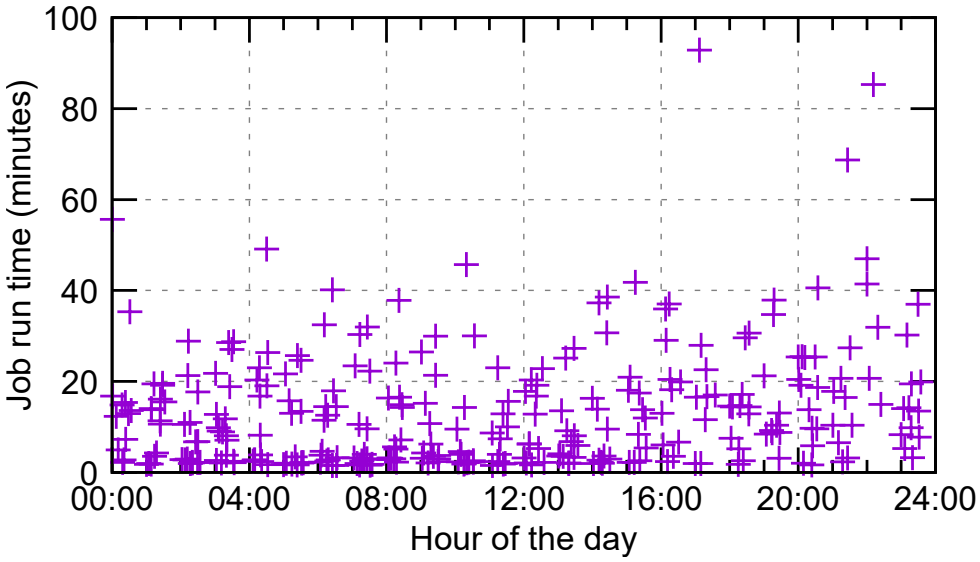


Fig. 4.1 Run times of 300 copy jobs over 2 weeks. All days are overlayed over their 24 hours.

Unfortunately, the prior works do not account for some important challenges and constraints of *real production systems* (Sections 4.2 and 4.7). For example, the server hardware in datacenters is heterogeneous in terms of resources and storage configurations (static heterogeneity). The performance isolation mechanisms of resource-harvesting datacenters produce another form of (dynamic) heterogeneity. Thus, tail latency management techniques evaluated in the absence of such static and dynamic heterogeneity may miss important effects. Datacenters are also very large; evaluating ideas on small systems side-steps many scalability challenges, *e.g.* difficulties with using centralized components. Perhaps most importantly, production systems must be simple and maintainable. Complex techniques (*e.g.*, relying on sophisticated performance modeling) are undesirable, as they require expertise and skills that most engineers are not trained on.

In this section, we focus on tail latencies in distributed file systems under these production constraints. In particular, we address the challenging scenario where the distributed file system only stores data for the batch workloads, but the latency-sensitive services have full priority over the shared resources (*e.g.*, CPUs, local disk bandwidth) on the same machines.

Thus, *we seek the best possible file system tail latency for the batch workloads;* service performance is protected by isolation techniques (*e.g.*, [57, 70]) and are *not* a topic of this work.

Though the batch jobs have more relaxed performance constraints than the services, lowering their tail latency is important because: (1) completing them faster frees up capacity for other jobs to run; (2) lower tail latency improves performance predictability and user satisfaction; and (3) batch jobs may be rendered useless if they take excessively long. As an illustration of the performance variation we observe, Figure 4.1 shows the run times of 300 executions of a simple I/O-bound job (distributed copy of a 200GB file) on 4k servers in a production datacenter over 2 weeks. As we can see, job performance may vary 60× or more, due to tail data access latencies.

These performance variations occur despite the fact that speculative execution (SE) is enabled in our jobs. SE tracks the tasks' durations and issues duplicate tasks for managing tail latencies in batch workloads [22, 23, 113]. Unfortunately, SE cannot tackle all sources of storage-level tail latencies without a stronger coupling between compute and storage layers [92]. Worse, SE may harm performance when resources are scarce [95, 83, 21, 92, 79], a common scenario in resource-harvesting datacenters. Thus, managing storage-level tail latencies independently is more attractive, leads to simpler systems (less coupling), and reduces the need for speculative tasks.

Along these lines, we first characterize the sources of storage-level tail latency impacting the run times of I/O-bound jobs shown in Figure 4.1 (Section 4.3). The characterization shows that data read and write accesses exhibit the most performance variation, not metadata operations. In addition, long disk queues and server-side throttling are the main culprits, not the utilization of the network, CPU or memory.

We then propose two client-side techniques for managing tail data access latencies: "fail fast" for writes, and "smart hedging" for reads (Section 4.4). Both techniques are oblivious to

the source of variations, and rely on simple server-side performance reporting and careful reactive policies, while leveraging the existing replication and fault-tolerance mechanisms in distributed file systems. Many systems use chain replication [98] for data writes, where each write request (possibly broken into chunks called "packets") is pipelined serially across the servers that store a replica of the block [12, 47, 13, 96, 24, 34, 45, 28]. In such systems, a slow server in the pipeline can significantly degrade write latency. However, prior works have almost always assumed that writes happen "in the background" (buffered writes). In contrast, our characterization shows that, in practice, several concurrent block writes on the same server can overflow the write buffer and significantly impact tail latencies.

To reduce data write tail latencies, our fail fast technique detects and replaces the slowest server in a pipeline. The new server must receive all packets that have already been written before writing can resume. However, aggressive server replacements may overload the system. In fact, since replacements are expensive, fail fast estimates the cost of replacing a server and performs a replacement only if it would indeed reduce write latencies overall.

Our smart hedging technique for data reads monitors the server's performance on a per-packet basis, and starts a "hedge" (duplicate) request [41] when performance starts to degrade. Since aggressive hedging may cause overload, we hedge adaptively and (exponentially) back-off from a server that does not complete a hedge before the original request.

To experimentally evaluate our techniques, we implement them in Hadoop Distributed File System (HDFS) [12], a popular file system for frameworks such as MapReduce [42] and Spark [27], and call the result "CurtailHDFS" (Section 4.5).

Our evaluation uses (1) synthetic workloads on a 4k-server testbed; and (2) production workloads in 3 datacenters with a total of 70k servers (Section 4.6). We compare CurtailHDFS to baselines that include typical server-side techniques for managing latency, such as tracking server performance. The results show that CurtailHDFS can reduce 99^{th}-percentile latency by 19$\times$ compared to HDFS for I/O-bound jobs. For the more balanced production workloads,

CurtailHDFS reduces the 99.9^{th}-percentile write latency by $2\times$ compared to HDFS, and the average read latency by $1.4\times$ compared to state-of-the-practice hedging. These are significant improvements, especially given the limited scope of our changes, on top of existing techniques (*e.g.*, speculative task execution).

Though we evaluate our techniques in highly heterogeneous resource-harvesting datacenters, they are general and applicable in other contexts as well. In fact, we are contributing our techniques to open-source HDFS, so they will immediately benefit frameworks and workloads that currently use it in any datacenter.

Our contributions are:

- We characterize the tail latency of batch jobs running on production HDFS in resource-harvesting datacenters.
- We propose general client-side techniques for managing tail latencies in distributed file systems, and implement them in HDFS. Our write technique uses an entirely new approach to shortening tail latencies, whereas our read technique improves on prior works.
- We evaluate our techniques extensively, and show that they lower tail latencies significantly.

4.2 Production challenges and constraints

There are many challenges to manage tail latency in production distributed file systems. In this section we enumerate several we encountered.

Massive scale. Datacenters host thousands of servers. At this scale, systems are more prone to transient misbehaviors or failures that cause latency tails. Moreover, they must avoid

centralized components. So, distributed file systems with a single primary metadata manager, such as HDFS, must be extended. Our production HDFS federates multiple HDFS sub-clusters by placing multiple software "routers" in front of the sub-cluster metadata managers, as proposed by Misra *et al.* [76].

High heterogeneity. Datacenter hardware is often heterogeneous, with at least a few server generations with different performance characteristics. Even servers of the same generation may use multiple types (SSD, HDD, or both) or configurations of storage media (*e.g.*, raided with different numbers of devices). There are 12 server configurations in the datacenters we study in this work.

Resource harvesting. Our production HDFS uses two isolation mechanisms to protect the co-located services: (1) throttling of the data access throughput of the server (at 60MB/sec), and (2) a "busy" flag that informs the corresponding metadata manager and clients that the server cannot take more access load. Distributed file systems that run on harvested resources may have their requests throttled or denied because of latency-sensitive service activity. We find that $\sim$20% of servers may concurrently become unavailable because of such activity.

Background replication. On server or disk failures, the lost replicas must be re-created in the background, which increases server load. The problem is exacerbated in our datacenters because the operators of the latency-sensitive services may reformat disks at any time. The reformatting rate varies in each datacenter, but in the worst case, up to 90% of servers get reformatted in a month [115]. To prevent data loss, our production HDFS stores the replicas of each block on servers that run different services, as proposed by Zhang *et al.* [115]. Despite doing this, it is common to have to re-create 25TB monthly, due to reformatting, with peaks of 128TB.

Load imbalance. At scale, data servers and/or metadata managers may become overloaded and provide poor performance. Our production HDFS deployments create an average of 2

blocks per second with spikes of more than 100. During these spikes, some servers are idle while others show more than 20 concurrent data accesses.

High hidden costs. Datacenter operators are careful to limit complexity in their systems, to lower labor and maintenance costs. Similarly, new system components and data pipelines are expensive to produce, operate, and maintain. Complexity and new infrastructure must produce significant benefits to justify their costs.

4.3 Characterizing Performance Variation

This section characterizes the performance variations we observe in real datacenters and their sources. We base our study on HDFS, but our observations directly apply to other well-known systems, such as Cosmos Store [38] and GFS [47], that have similar structure. We start with an overview of our production HDFS and our methodology. Then, we discuss the job/task and data access performance variations.

4.3.1 Methodology

Experimental setup. For our next experiments, we deploy our production HDFS on 4k servers across 4 sub-clusters in a resource-harvesting datacenter. The servers have 12-32 cores, 32-128GB of memory, 10-Gbps NICs, and a 6TB RAID consisting of 4 15k-RPM HDDs. In some servers, we also use a 1TB RAID of 2 SSDs to store the file system data. The NN in each sub-cluster replicates each block across 4 DNs in the same sub-cluster; replicas of a block may be stored on HDDs, SSDs or some combination of the two storage mediums.

Workload. Our experiment profiles a DistCP [15] job that spawns 200 tasks for copying a 200GB file between 2 sub-clusters. We run this experiment every 2 hours, for a week. Each run spans a few thousand servers; speculative task execution is enabled in each run. We use

TeraGen [16] to re-create the source file before each run. See Section 4.6 for our study of other workloads.

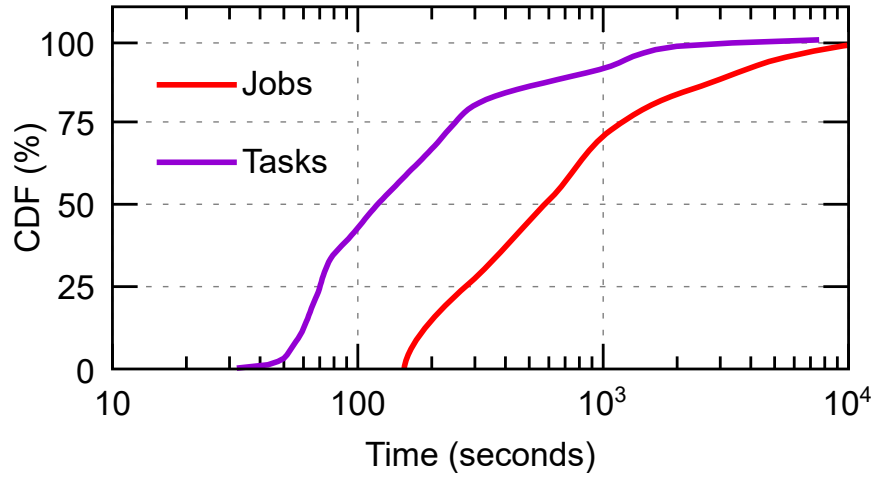
Monitoring infrastructure. We extended HTrace [14] to profile I/O operations at a packet granularity and collect system load metrics. During run time, each server logs details of profiled packets on a local disk (separate from HDFS data disk). Once a job finishes, a collection framework parses the job logs to determine, fetch and create request traces from the profile logs of servers involved in the job. We are contributing our extensions to open-source HDFS [18, 17].

4.3.2 Job/task performance variation

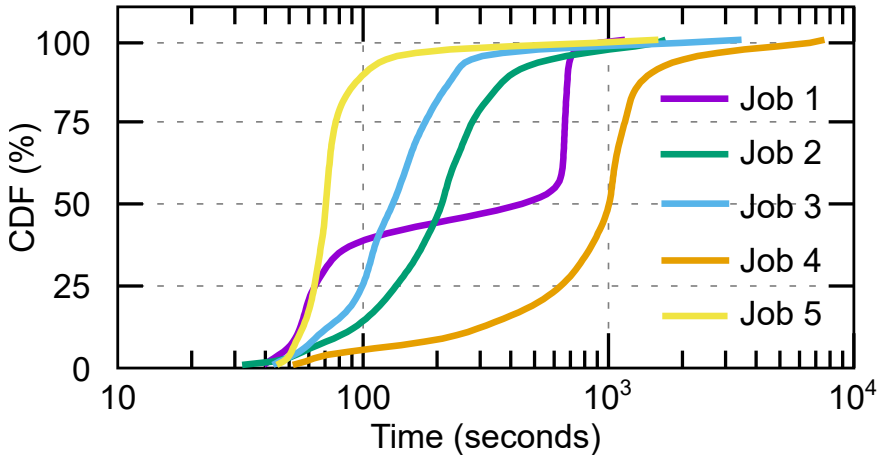
Figure 4.2a presents the CDF of the DistCP job runtimes. The average job run time is 20 minutes and the standard deviation is 27 minutes. More strikingly, *the fastest job takes 2 minutes whereas the* 99^{th} *percentile is 120 minutes.* To understand this huge variation, we consider the running times of tasks within a job.

Figure 4.2a also shows a CDF of the task runtimes across all DistCP jobs. Though each task does the same amount of work (copies 1GB), their runtimes vary significantly. The average task run time is 334 seconds and the standard deviation is 730 seconds. The fastest task takes 32 seconds whereas the 99^{th} percentile is $\sim$1 hour.

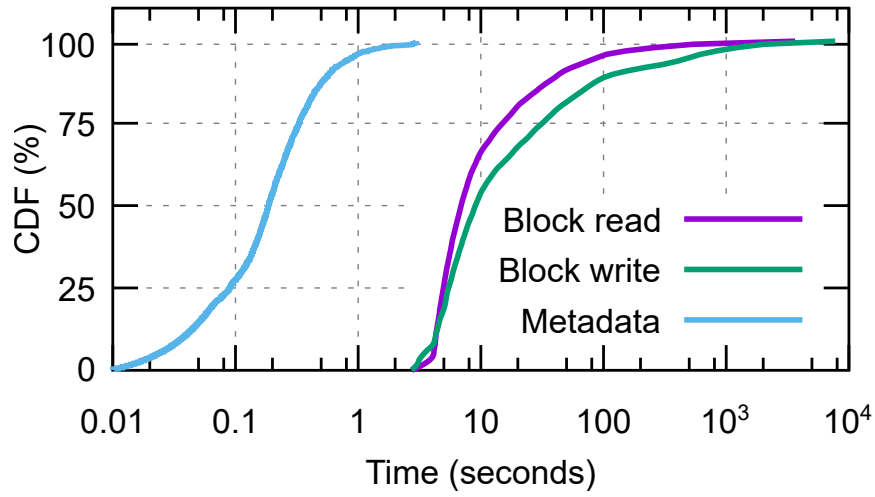
To explore whether there are common job slowdown patterns, we consider the task runtimes within a job. Figure 4.2b shows the results for a few DistCP jobs. Clearly, a few hotspots slow down jobs in some cases and heavy load on the entire system has an impact in others. For example, a few stragglers slow down Job 5. In contrast, more than 75% of the tasks are an order of magnitude slower than the fastest task in Job 4, indicating that the system is under heavy load. This is also corroborated by the fact that the fastest task in Jobs 4 and 5 has similar runtimes ($\sim$50 seconds), but the median task run time in Job 4 (1000 seconds) is



(a) Job and task running times.



(b) Task running times within sampled jobs.



(c) Latencies per operation type.

Fig. 4.2 Performance of DistCP jobs and file accesses.

Table 4.1 Correlating latency and DN resource utilization.

Resource type	Spearman coefficient (ρ)	
	Reads	Writes
CPU utilization (%)	0.23	0.31
Memory utilization (%)	0.29	0.25
Disk queue length	0.7	0.65
Network bandwidth (MB/s)	0.14	0.11

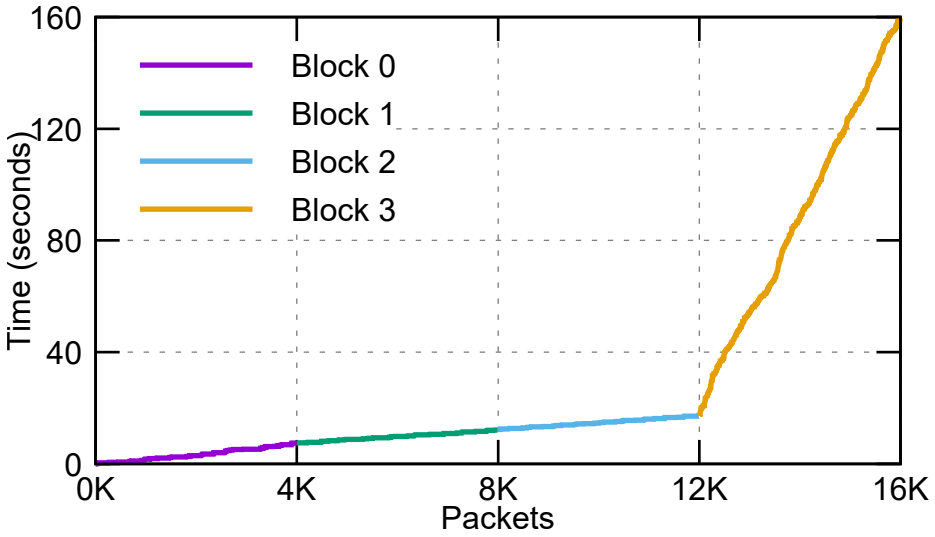
20× slower than the fastest task and 15× slower than the median task run time in Job 5 (70 seconds). The other jobs show similar behavior.

These extreme variations in runtimes are less likely to occur in other settings. For example, in smaller clusters lacking server heterogeneity [51, 92, 93, 110], or when experiments (*e.g.*, background load) are tightly controlled. In contrast, datacenters exhibit more static and dynamic heterogeneity, non-uniform server load distributions [57], interference from co-located applications, and server re-imaging/restart by cluster management systems.

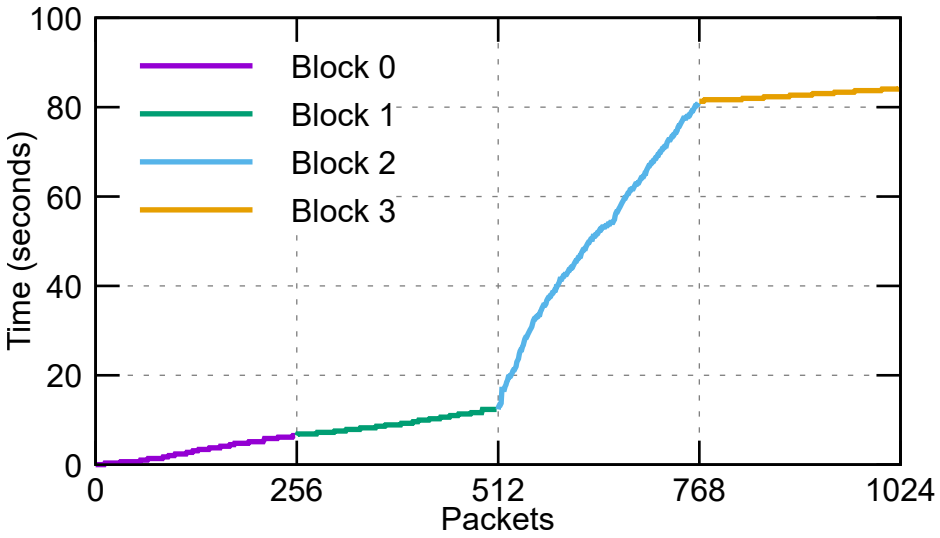
4.3.3 Sources of performance variation

In this section, we analyze the traces of slow tasks to determine operations that cause tasks to slowdown. Figure 4.2c shows a CDF of the operation latencies. We classify operations into block read, block write and block metadata. Reads and writes suffer major slowdowns: the 99^{th} percentile latencies are an order of magnitude greater than the median, but metadata operations have little impact since the 99^{th} percentile metadata operation latency is similar to the fastest reads and writes. Note that using sub-cluster federation already reduces the performance degradation in the metadata operations.

Next, we look at individual traces of writes and reads to determine the causes for slowdown.



(a) Write four 256MB blocks to a 1GB file.



(b) Read four 256MB blocks from a 1GB file.

Fig. 4.3 Sample write and read timelines. Each color represents a block. Each point depicts a set of bytes transferred during a write or read. Time is on the Y axis.

Reads. We correlate between block read latencies and DN resource utilization in Table 4.1. Our correlation analysis shows that disk contention (either from latency-sensitive services or other batch jobs) and the resulting queuing delays are the primary cause for performance

variation. Figure 4.3b shows the packet latencies while reading four blocks. As we can see, block 2 takes significantly longer (more than 1 minute) to read. However, long disk queues are not the only reason. Another factor is server-side throttling that limits the overall DN throughput at 60MB/s (instead of the many hundreds of MB/s we can get from the RAIDed storage devices), regardless of how many clients are accessing the server. *This effect would not have occurred in dedicated clusters or other setups where there is no resource harvesting.*

Writes.[1] Like reads, our correlation analysis between block write latencies and DN resource utilization (Table 4.1) indicates that disk contention has a substantial impact on performance variation. Figure 4.3a illustrates the write performance variation. It shows the observed latencies while writing the packets of the four blocks in a file, each represented with a different color. Block 3 suffers a huge slowdown because of high write latencies through its pipeline, taking more than 2 minutes to complete. Two servers exhibit large disk queues during the writing of this block.

Other sources. We find cases where contention for other resources (*e.g.*, CPU, network bandwidth) has an impact on performance, but they are rare.

4.3.4 Summary

We find that:

1. Job running times are highly variable and without clear slowdown patterns, such as time of day.
2. Load from both latency-critical services and other batch jobs impacts performance.
3. Data read and write accesses exhibit the most performance variation, not metadata operations.

[1] Note that writes are not targeted in this thesis. However, they are part of of the larger scope of the project [77].

4. Long disk queues and server-side throttling introduce the most variation, not the utilization of the network, CPU or memory.

These effects are more likely to occur in large production systems.

As Figures 4.3a and 4.3b show, long disk queues and throttling can last for minutes, whereas other block accesses take seconds. In addition, although not seen in our characterization study, contention for other resources can cause similar performance variation for data accesses, *e.g.*, contention for CPU in erasure-coded systems and/or due to high demand from co-located CPU-bound applications. This means that, *regardless of the source of the delay,* it is important to react, *i.e.* direct accesses away from these slow servers. However, we need to react carefully so (1) transient effects do not cause unneeded reactions; (2) older, less-performant servers can still be used; and (3) we avoid reactions when all servers are overloaded.

4.4 Managing read tail latencies

In this section, we present our technique for tackling the sources of performance variability in read accesses from Section 4.3. Throughout the section, we use the general terms "metadata server" and "block server", as our techniques apply to distributed file systems beyond just HDFS.

4.4.1 Basic principles

Our technique leverages information available at the distributed file system client library to make decisions. At a high level, it proceeds through the following phases: *track latency*, *check for slowness*, *decide whether to act*, and *take action*. In the *track* phase, the client stores the packet and block latencies for each block server with which it is communicating.

It then uses these tracked latencies in the *check* phase to determine whether a server is slow. This check needs to be agnostic of the source of the delay and, simultaneously, loose enough to avoid reacting to transient conditions or older, less-performant servers. The client runs the check phase for the first time after it has collected enough latencies to make a meaningful decision. If the check determines that a server is slow, the client *decides* whether to take a mitigation action (*i.e.*, hedge for a read). This decision includes backing off from overloaded servers (reads). If the client decides that it will be beneficial to act, it *takes the action* and goes back to the tracking phase.

Our approach adapts to dynamic load changes by only considering recent requests in the check phase (load may have changed recently at the servers), and by comparing the server's request latency against those of other servers (all servers may be highly loaded). Moreover, the approach is agnostic to the reason for slowness, *e.g.* delays due to other file system requests, latency-sensitive service activity, or throttling.

Importantly, these basic principles conform to the constraints of production environments. First, they enable a client to make local decisions and do not require any centralized infrastructure in the decision-making process. Therefore, they do not pose any impediment to scalability. Second, they adapt to dynamic changes and do not make any assumptions about the system, which makes them amenable to run on harvested and heterogeneous resources. Third, they can be easily deployed and integrated into existing systems, since our techniques are simple and do not require invasive server-side changes.

Below, we detail our technique and discuss the alternate approaches we considered in Section 4.4.3.

4.4.2 Smart hedging for reads

Request hedging [41] leverages the multiple copies of data for cutting read tail latencies. A popular approach is to issue a hedge request, if the latency of the current request is

greater than a static threshold. Figure 4.4 illustrates read hedging, where a client switches to another replica (server 2) after observing a high read latency (from server 1). Although this static approach can cut tail latencies, its effectiveness depends on the value of the threshold: an excessively high value may result in infrequent hedging, whereas an excessively low value could overload the system. We actually observed the latter behavior in production (Section 4.6.4). Moreover, even a finely-tuned threshold value may require readjustment under varying load conditions (dynamic heterogeneity).

Thus, our smart hedging technique uses a *per-client* dynamic threshold along with a retry policy to control the hedging rate. Each client constantly records the latency of the packets it reads from servers (track). Initially, it uses a static threshold to trigger hedging and over time leverages the tracked latencies to adjust the threshold (check). The check compares the latency of the server against a multiple of a high percentile of the latencies the client has experienced from other servers. Note that the client tracks the latency on a server basis, but keeps a dynamic threshold for all servers. In this case, the multiplicative factor compensates for variability in request service times [93] or queuing delays [52]. (In our HDFS implementation, the default high percentile is the 95^{th} and the multiplicative factor is $3\times$.) As the value of this high percentile latency changes, the hedging threshold for each client changes as well.

If a request is taking too long, the client considers sending a hedge request to the next server on the replica list for the block (decide), say server 2. Before issuing the hedge, it checks whether the last hedge to server 2 completed after the corresponding non-hedge request (*i.e.*, the last hedge to server 2 was unsuccessful). If so, it exponentially backs off from server 2 and considers the next one on the list. Otherwise, it issues the hedge to server 2 (action). Exponential backoff ensures that we do not attempt to access an already overloaded server, and works well in practice. Specifically, our smart client waits for an exponentially increasing number of packets before re-attempting a hedge request to a server who failed a hedge in the past attempt.

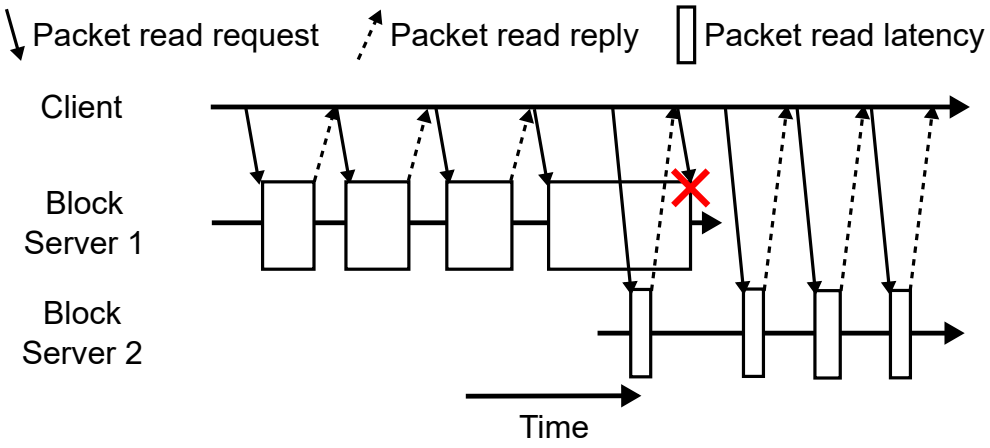


Fig. 4.4 Read hedging.

4.4.3 Alternative techniques we discarded

Centralized controller. Instead of making independent, local decisions about contention, the clients could access this information in a central repository maintained by the metadata manager. However, this approach would be highly dependent on the freshness of the load estimations. If the estimation window is too large, the manager would provide stale data and may not reduce tail latencies. On the other hand, a small window could provide accurate information but may overwhelm the manager. As others have observed in production [76], centralized components often harm scalability.

A different alternative would be to introduce a different centralized structure in the distributed file system. However, this introduces additional network overhead to coordinate between clients and the centralized structure, as well as a centralized storage point to keep track of relevant measurements such as latency. Our approach relies on local data, thus we avoid coordination overhead between clients and centralized structures.

OS-based improvements. A design goal of many large-scale distributed systems, such as HDFS or Spark, is to provide portability across platforms. Thus, we focus on improving

file system performance in the user space, rather than modifying the underlying software or requiring features that few operating systems provide.

4.5 Implementation in HDFS

We implement our technique in HDFS, together with fail fast writes (technique to mitigate tail latency for write operations [77], we call the resulting system "CurtailHDFS".

We create a new client library that applications can use. The library allows each technique to be enabled independently, which is key for assessing their benefits in production. Moreover, adoption can be incremental as users slowly deploy our client.

Our client keeps a configurable amount of the recent history of packet and block latencies. For smart hedging, we extend HDFS's static hedging option [106, 107]. with a per-client dynamic threshold and exponential backoff.

We avoid using any DN that has flagged itself as "busy" (a latency-sensitive service needs the server's resources). We discuss the impact of the techniques' parameters in Section 4.6.2.

Finally, our code is modular and easy to add/remove/configure, as needed for maintainability in production.

4.6 Evaluation

In this section, we describe our experimental methodology and results. We conclude with a discussion of our experience bringing Smart Hedging, as part of CurtailHDFS[2], to production.

[2]The CurtailHDFS contains solutions to mitigate tail latency of read and write operations, smart hedging and fail fast, respectively. Fail fast is not a topic of discussion in this thesis, but more information can be found in Misra *et al.* [77].

4.6.1 Methodology

Experimental setup. We evaluate CurtailHDFS in two environments: the 4k-server experimental testbed we describe in Section 4.3.1; and 3 large production deployments, each with 20k to 30k servers spread across 6 to 10 federated sub-clusters. These deployments use the same server configurations as the testbed, but exhibit more sharing with latency-sensitive services. The replication factor is 4 in the testbed and production deployments.

Evaluating our techniques in production deployments is difficult, because both the batch and latency-sensitive service loads are constantly changing. To do so accurately, we take an "A/B testing" approach where we run multiple techniques simultaneously. Specifically, we configure some clients to run the technique(s) we are evaluating while other clients run the baseline(s) for comparison. To simplify log processing, all clients on the same machine run the same technique(s). In addition, the deployments experience periods of relative inactivity that are uninteresting. Thus, we collect file system telemetry in 10-minute periods, and focus our evaluation on those periods that experience both more than 1000 full-block reads and 1000 full-block writes. These are the minimum number of accesses we need to compute 99.9^{th} percentiles for each access type. Table 4.2 lists the percentage of time these intervals represent.

Workloads. We evaluate CurtailHDFS with synthetic and production workloads. For our synthetic workload, we extend DistCP in three ways. First, we allow one of the sides of the copy to be main memory, so that we can either only read file data or only write file data. Second, we allow the setting of the number of tasks that should use each technique (*e.g.*, 1000 map tasks should read blocks using smart hedging and 1000 map tasks should use static hedging). Third, we instrument the code to collect key metrics from the client, such as latency and number of hedges. We call the resulting program ExtendedCP. We also implement a MapReduce job that creates a pre-defined amount of background file system load (*i.e.*, amount of HDFS open connections) in our testbed. The job monitors the load in

Table 4.2 Characteristics of production deployments.

Metric	DC0	DC1	DC2
#servers	25.2k	32.4k	19.0k
#server configs	4	4	4
#local storage configs	3	3	3
%servers shared with services	66.5%	29.4%	28.7%
%time in 10-min intervals	29.2%	72.0%	36.0%
#reads	2.6M	43.3M	38.6M
#writes	2.3M	8.3M	9.0M
#compute frameworks	4	4	4

the cluster and generates/terminates read/write operations to maintain a constant load over time. Moreover, we ensure that the distribution of load it generates across DNs is the same as in the production workloads.

In contrast, the production workloads are organic to our deployments and come from multiple engineering teams. They comprise various data analytics frameworks and applications. Spark and machine learning training, respectively, being the most common. The number of tasks in each job varies from a few to tens of thousands, whereas the tasks' lifespans vary from a few seconds to hours. Another common application is moving data between systems (*e.g.*, Cosmos to HDFS) for other applications (*e.g.*, training of machine learning models) to use. Section 4.6.3 details the patterns of these workloads. A large percentage ($> 90\%$) of the jobs access the distributed file system with our extended client, but others still use older clients.

Monitoring infrastructure. In Section 4.3.1, we describe our HTrace-based fine-grained monitoring infrastructure. The data provided by this infrastructure is extremely detailed, which leads to high collection, aggregation and processing times. So, it is only suitable for characterization and troubleshooting. Our synthetic and production workloads do not need this much detail. For our synthetic workloads, we use the metrics we collect from ExtendedCP. For the production workload, we extend the client to log per-block statistics (*i.e.*, total bytes, time of the operation, and throughput) for each read/write operation [18].

We rely on the existing Hadoop infrastructure to collect and store these logs in a repository (in HDFS). To speed up our analysis process, we implement a MapReduce job that parses all application logs and gets statistics related to (1) the throughput of each access, (2) hedges (tried and successful), and (3) fail fast replacements. In production, we parse these logs, aggregate the data, and expose the statistics as performance counters.

4.6.2 Synthetic workload results

We first evaluate CurtailHDFS with our ExtendedCP job in the 4k-server experimental testbed.

Reads. We now evaluate smart hedging against the baseline and the baseline with static hedging (100ms threshold), using the same setup as above but reading 1GB files. Without background file system load, the medians of the 99^{th}-percentile read latencies are 36 seconds for the baseline, 22 seconds for static hedging, and 32 seconds for smart hedging. The static version triggers a median of 56 hedges whereas the smart approach uses only 22. This trade-off becomes starker as the load increases. For instance, during high load scenarios, common in production environments, additional (and unnecessary) load can generate a cluster overload; leading to job failures and under-utilized resources, among other consequences. We provide a concrete production experience in Section 4.6.4. We run 10 more experiments with 30k clients generating high background load by reading from HDFS. (Reads are less expensive than writes, so they need more clients for the same amount of load.) In this case, the medians of the 99^{th}-percentile latencies are more than 22 minutes for the baseline, 81 seconds for static hedging, and 89 seconds for smart hedging. In addition, static hedging triggers a median of 97 hedges whereas smart triggers only 21. Figure 4.5 shows the latency (top) and the hedging (bottom) statistics for one of these experiments. In the hedging graph, the vertical ranges go from the minimum to the maximum number of total and successful hedges, whereas the blue boxes range from the 25^{th} to the 75^{th} percentiles. The horizontal line across the boxes is the median value. The figure shows that smart hedging is effective at

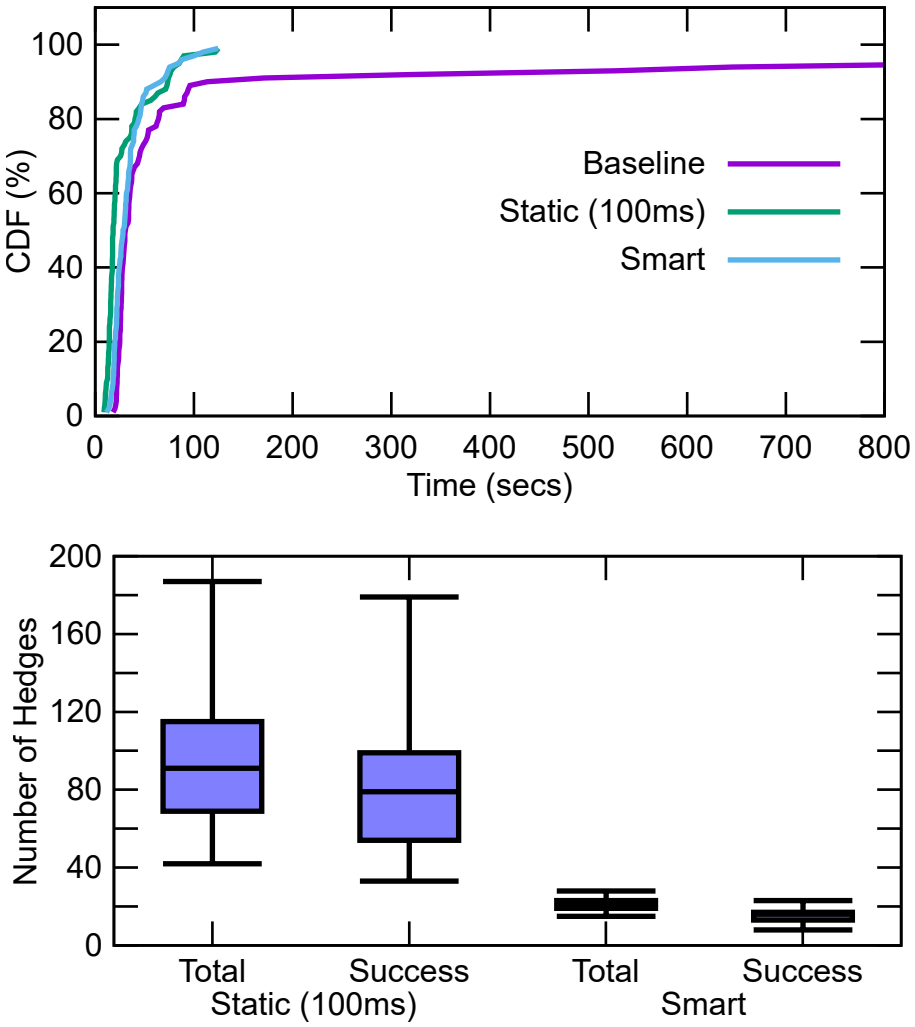


Fig. 4.5 Latency (top) and hedges (bottom) of 1000 reads under heavy load.

lowering the latency with a much lower number of hedges. Overall, the tail latency for static and smart hedging is similar at higher loads, while the number of hedges is much smaller under smart hedging.

Static hedging depends on a single parameter: the time to wait for a read before triggering a hedge. This threshold can be too conservative thereby offering little benefit, or too aggressive thereby increasing the load on the system. For example, we have observed scenarios where, under heavy load, the common-case read latency was greater than the timeout in the static technique. In such cases, the clients kept bouncing between replicas.

In general, we observe that static hedging thresholds longer than 100ms degrade latency, especially under low background load. However, in a production environment with many co-located latency-sensitive services, it makes sense to use a higher value (*e.g.*, 500ms as in our deployments) to reduce the amount of hedging load. Smart hedging does not depend strongly on this parameter as it only uses the static value for bootstrapping.

Reads sensitivity analysis. Smart hedging has the same three parameters as fail fast, as well as the retry policy. Higher percentiles and higher slowness factors do not impact latency significantly, but do reduce the number of hedges. As for writes, the best trade-off is to use the 95^{th} percentile and a factor of $3\times$. The number of recent requests to consider does not have a significant impact and medium numbers (*e.g.*, 20 samples) provide a good trade-off. Finally, the retry policy has a large impact on both latency and number of hedges. The best policy is exponential with 3 retries, which provides a latency comparable to static hedging with fewer hedges. Thus, it provides low latency without excessively increasing the system load. In contrast, a highly restrictive retry policy (*e.g.*, never retrying a failed hedge) reduces significantly the number of hedges, but latency increases substantially.

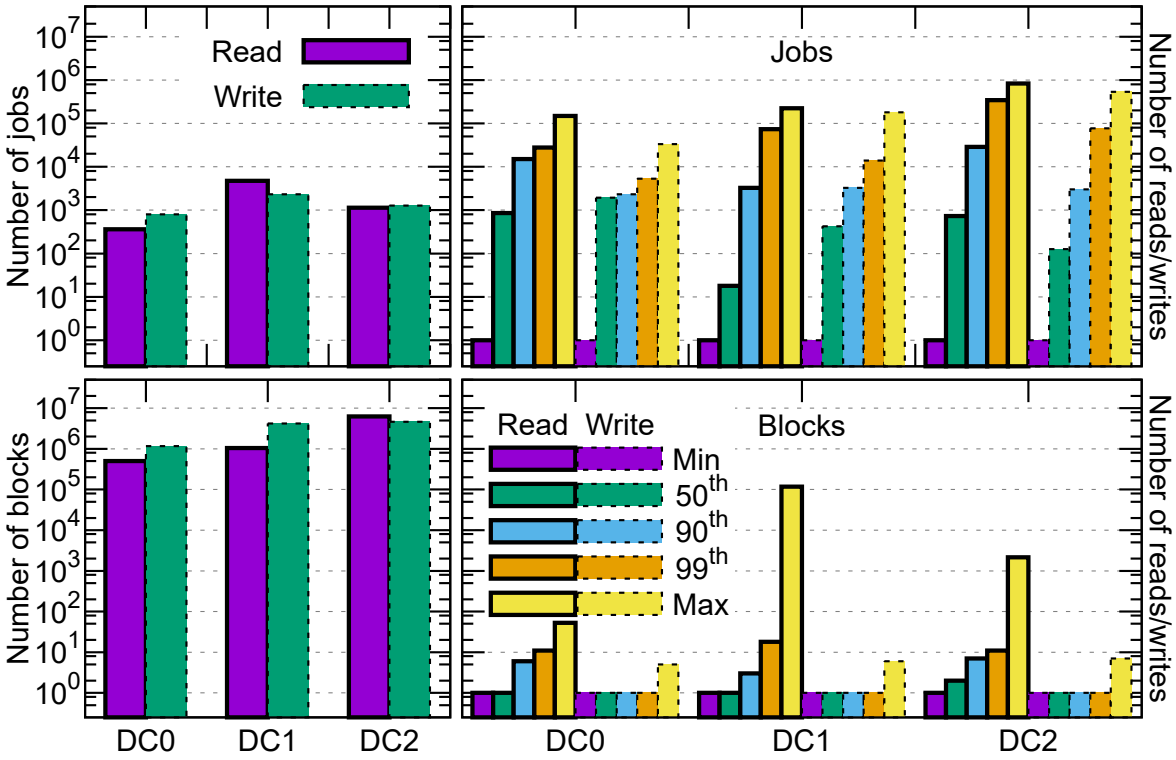


Fig. 4.6 Characteristics of production workloads. Note the same log scales in the left and right Y-axes.

4.6.3 Production results

Table 4.2 lists the main characteristics of our production deployments and workloads for one month. The workload data includes only the 10-minute intervals for which we report results. Recall that our evaluation focuses on time intervals with enough file system load to meaningfully compute 99.9^{th}-percentile latencies. Most rows in the table are self-explanatory. The three local storage configurations are SSD RAID only, HDD RAID only, and both SSD and HDD RAIDs. The "%time in 10-min intervals" row lists the percentage of the month that we are reporting about in the 10-minute intervals. The four compute frameworks include Spark and MapReduce. As we can see from the table, our deployments are large and heterogeneous in terms of hardware and software.

Figure 4.6 illustrates the characteristics of the production workloads (again only accounting for the 10-minute intervals) in each deployment. The graphs on the left show the numbers of jobs (top) and blocks accessed (bottom) in log scale. We split the job data into jobs with read accesses and write accesses; the two sets overlap, but not every job reads data and not every job writes data in our intervals. The graphs on the right show the distribution of reads and writes across jobs (top) and blocks (bottom) in log scale. For example, the leftmost bar in the top right graph shows the number of reads that the job with the minimum number of reads performed in DC0. Similarly, the leftmost bar in the bottom right shows the minimum number of reads that a block received in DC0. On both sides, the bars with solid and dashed contours represent reads and writes, respectively.

Unlike our synthetic jobs, the figure shows significant skew in the distributions of the number of accesses across jobs, and even greater skew in the distribution of the popularity of blocks for reading. Blocks only get written more than once when the first write does not write the entire block. Our intervals include any access to 256MB or more, but in some cases users define their blocks to be larger than 256MB. That is why the maximum number of writes per block is more than 2 in our deployments.

Although we do not illustrate this, the production workloads are more balanced than our ExtendedCP job, using much more compute cycles per file system access.

Figure 4.7 shows the aggregate client write and read access throughputs (in MB/second) in our intervals. In all deployments, throughputs vary significantly (up to 3 orders of magnitude) over time, and read throughput tends to be higher than write throughputs.

In the context of these deployments, Figure 4.8 summarizes the latency results in *log scale* for reads across many percentiles; the leftmost bars show the average results. The bars compare smart hedging against static hedging (we cannot use regular reads in production, as users are already accustomed to the lower tail latency of hedging). The height of the bars is the average value for the percentile/average across the 10-minute intervals, whereas the

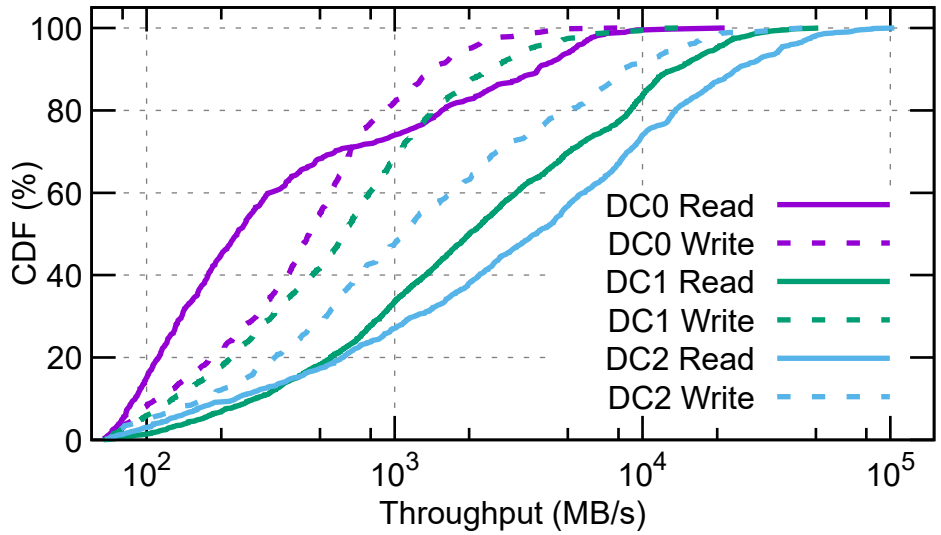


Fig. 4.7 Throughput in the production deployments.

vertical ranges go from the 5^{th} to the 95^{th} percentiles of the distributions of the 10-minute intervals.

The results show that smart hedging reduces latency in the middle percentiles (25^{th} to 95^{th}), as we are comparing it to static hedging configured for low traffic with a threshold of 500ms. (As we mention in Section 4.6.4, the initial setting of 100ms for static hedging was generating too much traffic in production, so it had to be increased to 500ms.) For these percentiles, the improvements to the low end of the distributions (bottom end of the vertical ranges) is particularly pronounced. *On average, smart hedging reduces latency by* 1.4× *compared to static hedging.* The improvement difference between synthetic and production workloads is due to two factors. (1) The higher load present in production environments, this includes the fact that static hedging has a threshold of 500ms, rather than 100ms as in synthetic evaluations. (2) The different workloads (and their resource usage), as illustrated in Fig. 4.6. In particular, the synthetic workloads were more I/O intensive, while the production workloads have a more balanced use of different resources (*e.g.*, network, CPU).

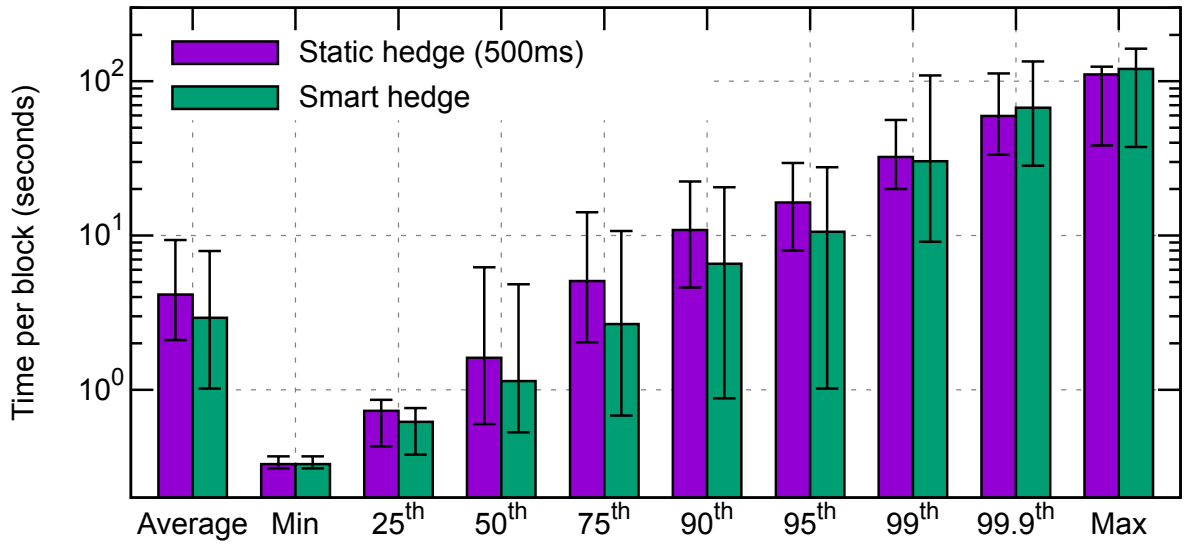


Fig. 4.8 Summary of production results for a month. Note the log scales in the Y-axes.

These improvements are remarkable for two main reasons: (1) our baselines already include techniques that help mitigate tail latencies (*e.g.*, speculative task execution, static hedging, servers warn clients and metadata managers when they are busy); and (2) they embody all challenges of production systems (*e.g.*, need to be simple, high heterogeneity, extreme scale, real workloads, interference from services).

To understand these results at a deeper level, we next detail the impact of our techniques for writes and reads.

Reads. Figure 4.9 shows the distribution of the average read latencies (left) and the distribution of the average number of hedges per block request (right), across the 10-minute intervals. Clearly, the average read latency is lower in nearly all intervals. Hence, smart hedging reduces tail latencies without negatively impacting the average read latency.

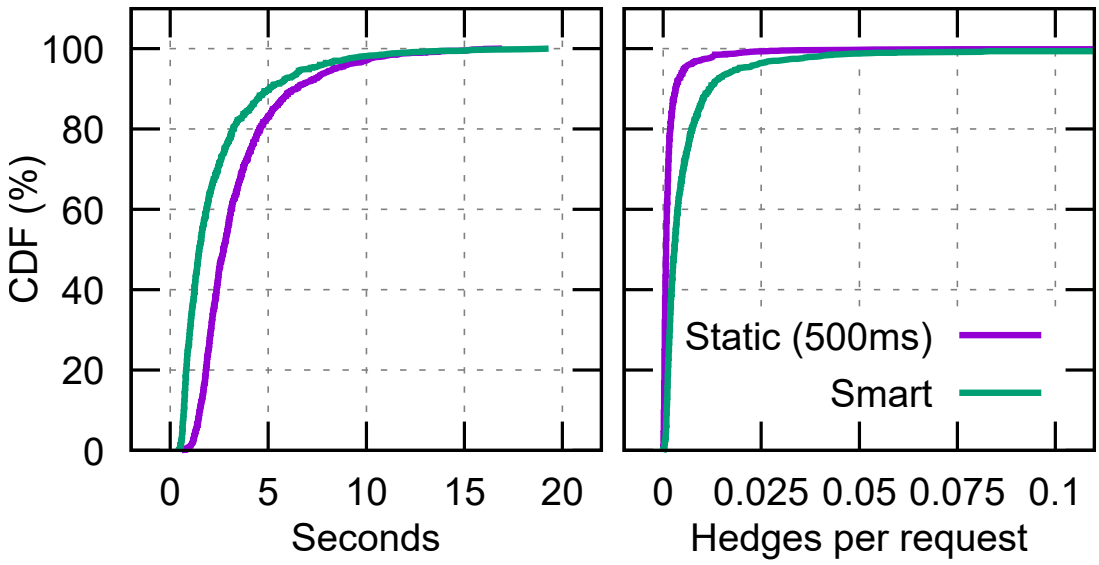


Fig. 4.9 Average read latency (left) and number of hedges per block (right).

Figure 4.9 also shows a side-effect of smart hedging: slightly more hedges per request on average compared to static hedging. The static hedging threshold in production deployments is 500ms, up from the 100ms threshold in the 4k-server testbed (Section 4.6.2). A higher threshold in production reduces the number of hedges, which is especially important under heavy load (Section 4.6.4). In contrast, smart hedging uses the 500ms threshold only for bootstrapping and later adapts the threshold based on observed read latencies. This approach reduces tail latencies, but can also cause more hedges in scenarios where the load across replicas is non-uniform (*e.g.*, high for a few and low across the remaining replicas).

Coming full circle. We motivated this chapter by showing that I/O-bound jobs using our production HDFS (plus static hedging) exhibit large performance variations in one of our production deployments (Figure 4.1). Figure 4.10 shows the same data, and CurtailHDFS results for these jobs in the same deployment; speculative task execution is enabled in jobs on both systems. CurtailHDFS significantly improves performance across most of the spectrum, but especially in the high percentiles (by $\sim 3\times$).

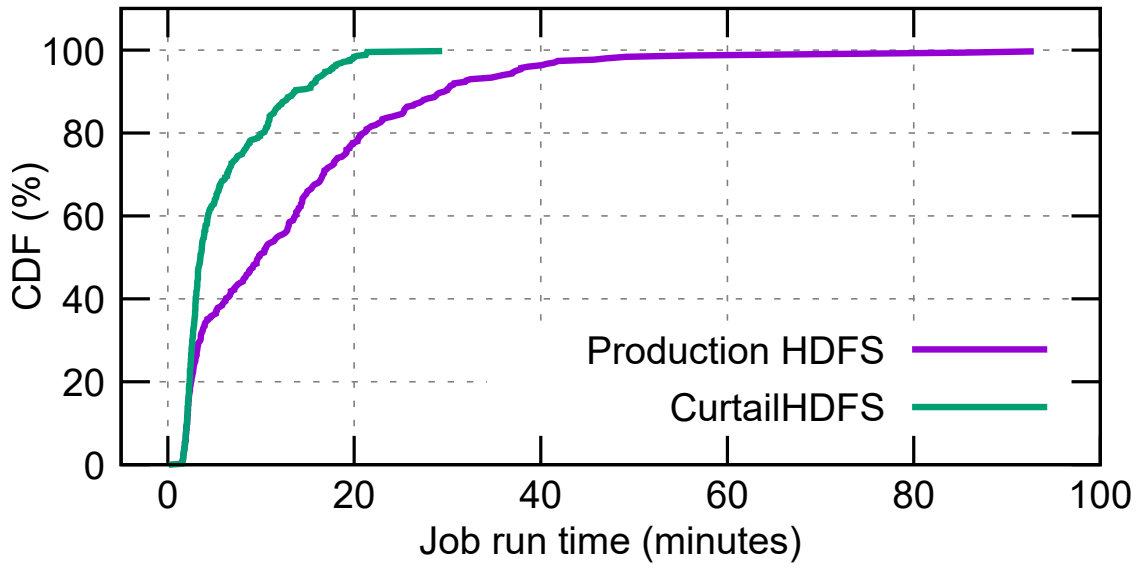


Fig. 4.10 CDF of copy job run times over 2 weeks.

Unfortunately, we cannot perform the same analysis with production jobs, as we do not know whether two executions of a job use the same data (we expect that they do not) or even the same code.

4.6.4 Experiences in production

Our experience taking CurtailHDFS from the 4k-server testbed to production illustrates issues that are often overlooked in the literature. We first experimented extensively with I/O-bound jobs in the testbed, always observing significant improvements from our techniques.

We deployed static hedging in production in one datacenter with a threshold of 100ms. Read latency improved significantly and we enabled it in all deployments after 3 months. However, after a few months, we started to observe periods when some servers were becoming overloaded. After some investigation, we realized that this behavior was caused by the high number of hedges under high load. To mitigate this issue, we increased the threshold to

500ms. This significantly reduced the load, but it also reduced the effectiveness of the technique. As a result, tail latencies were still reduced significantly, but the average latency increased. Clearly, static hedging is heavily dependent on its threshold and load. So, we introduced smart hedging to reduce read tail latencies, without negatively impacting the average latency.

We concluded the A/B testing and enabled our tail-mitigation technique by default in the modified HDFS client. The technique has improved the user experience significantly and many users have already switched to our client.

4.7 Related Work

Tail Latency in Distributed File and Storage Systems. Several works addressed read tail latencies [93, 81, 37, 106, 107, 96, 105]. C3 [93] proposes a replica selection algorithm that uses server-side information and rate limits. However, C3 is not scalable as it needs to store server load data on ephemeral clients. Rein [81] targets key-value stores and schedules concurrent reads of multiple keys. CRAQ [96] improves read throughput and latency under chain replication by enabling reading from any replica in the chain, while still providing strong consistency. Smart hedging is orthogonal to CRAQ and can reduce its read tail latencies. Other works propose techniques that run at pre-defined intervals. For example, Cassandra uses dynamic snitching [37], which periodically ranks servers based on observed latencies and server-reported loads to select the best replica for a read. CosTLO [105] issues redundant reads and writes in key-value stores, and considers the first response. CosTLO searches through several configurations, estimates the latency cost of particular configurations, and picks a cost-effective configuration. HDFS has a (static) read hedging option [106, 107] that starts a duplicate (hedge) request to another server, if the original request has taken longer than a threshold. Unfortunately, these techniques cannot tackle dynamic heterogeneity

since the interval/threshold might be inappropriate at run time. In smart hedging, each client determines when to react at run time, and uses back-off to prevent overload.

Earlier works [49, 60, 88] manage resources to meet tail latency SLOs and ensure fair sharing. For example, Cake [102] and Avatar [114] use a centralized reactive feedback-control scheduler for proportional sharing. PriorityMeister [116] uses proactive techniques for bursty workloads, workload priorities, and rate limiters. SCADS [97] uses model-predictive control for resource allocation. These works are orthogonal to our techniques, and often rely on complex performance models.

Tail Latency in the Storage Stack. Many distributed file systems and storage systems rely on layering, rather than end-to-end platforms, to provide storage functionality. Therefore, performance degradations in lower layers of the storage stack (*e.g.*, local file systems, block layer, storage devices) can negatively impact the performance of upper level storage systems. Hao *et al.* [52] characterize tail latency of storage device. Their study finds that tail latency is mainly caused by internal characteristics of storage devices and propose tail-tolerant RAID. Tail-tolerant RAID endures tail latency in read workloads by using an analogous approach to hedging. Chopper [55] analyzes the impact of file system policies on performance behavior. Chopper shows the impact of block allocation policies and poor layouts on tail latency. MittOS [51] provides operating system support to reduce tail latency and achieve Server Level Objectives (SLOs). In MittOS, the operating system exposes SLO interfaces in I/O requests to user-space applications. The SLO interface allows the operating system to rapidly reject I/O requests that cannot be served in time. When an I/O request is rejected, it is sent to another node to be handled. MittOS techniques are complementary to our solution.

Tail Latency in Data Analytics Frameworks. A large majority of data analytic frameworks rely on speculative execution (SE) to mitigate tail latency. LATE [113] is a scheduler for Map-Reduce frameworks. LATE speculatively executes the task (within a job) that is expected to finish further in the future. Mantri [23] uses real-time progress reports to monitor, detect, and act against straggler tasks using resource aware techniques. Dolly [22] fully clones *small*

jobs, avoiding the waiting time in previous speculative execution solutions. Dolly relies on the fact that small jobs consume a little amount of resources; thus cloning full jobs increase resource utilization marginally. Unfortunately, this approach does not work for large jobs, in which clones may significantly increase cluster resource utilization. These techniques improve running times by cutting compute bottlenecks. However, they cannot manage storage bottlenecks. PBSE [92] couples compute and storage, to allow better detection of stragglers by exposing dataflows characteristics to the compute framework. Our technique handles storage-level bottlenecks independently from computation, while avoiding greater compute-storage coupling.

4.8 Summary

We introduced *smart hedging*, a technique to manage read tail latencies in production distributed file systems, distributed file systems. We implemented it in a popular system, and evaluated it extensively. Our results demonstrate large latency improvements with I/O-bound jobs, and smaller improvements with more balanced production workloads. The results also illustrate that it is easy to overlook important effects in non-production systems. We conclude that it is possible to devise effective techniques while considering the challenges and constraints of real datacenters.

Chapter 5

Future Work

In this chapter we propose different ideas in which this thesis can be extended.

Storage devices, block layers, and local file systems are fundamental infrastructure components in today's computer systems. The better understanding of the interaction between multiple components, as well as tunable parameters, would increase performance stability across all systems in the storage stack. Extending our study could be a first step to achieve this goal. For instance, we could analyze the interaction between different file systems (*e.g.*F2FS [66], XFS [94], BetrFS [59]) and SDDs; as well as how different I/O schedulers impact performance variability. Our analysis solely focused on sequential read workloads; it would be interesting to see if the slowdown patterns observed there are also present in random read workloads, sequential write workloads, random write workloads, and mixed workloads. One more possible extension to this work consists of using the analysis insights to develop local file systems that are more resilient to performance variability or that can at least automatically detect it.

Another option is to develop applications that are able to detect performance degradation due to lower layer misbehaviors. For instance, by modifying the local file system APIs, applications can automatically detect slowdowns and react accordingly. This is somewhat similar to what MittOS proposed with `read` system calls [51]. This approach could be extended across upper layers of the stack and across other system calls. With these changes,

applications can develop policies to provide performance guarantees. However, this approach would require API modifications on each one of the lower layers in the stack. Unfortunately, this would affect portability, one of the key benefits that many of these systems target [87], as well as maintainability, and ease of deployment.

Chapter 6

Conclusions

Over the last years, large amounts of data demand efficient computer systems. In particular, efficient storage systems, since they are in charge of storing and retrieving the data. To keep up with the demand, storage systems have evolved quickly across the stack; from new storage media (*e.g.*, PCIe SSDs, SMR disks), local file systems (*e.g.*, F2FS [66], BetrFS [59]), to layered storage systems (*e.g.*, distributed file systems, key-value stores). In this thesis, we studied and improved the performance of storage systems.

We first analyzed the interaction of SSDs and ext4 in sequential read workloads, with a focused on buffered I/O. In our study, we performed the same experiment over extended periods of time, and monitored the I/O behavior with existing Linux tools. The experiment consisted on reading a large file (*i.e.*, 256MB), both with HDFS and FIO, using system calls that performed buffered I/O. We found that throughput varied differently depending on the file lifetime and SSD characteristics. For instance, the performance of read operations varied regularly in the beginning of a read operation that concerned a file fragment. Moreover, a file with a long lifetime (*i.e.*, weeks) experienced a permanent slowdown. Overall, we uncovered three performance instabilities, we refer to them as intrinsic, temporal, and permanent slowdown. When combined, these slowdowns can decrease throughput up to 30%. We determined that the primary causes of these variations are the interaction between SSD misbehaviors and lack of enough parallelism from the local file system. We showed that the local file system can mitigate these variations by requesting larger and more I/O requests.

However, in buffered I/O applications this cannot be direct mitigation, since larger requests would fill the buffer cache faster, which might remove the benefits of using a buffered cache. Our analysis points to the fact that local file system need to become more adaptable, specially as multiple parts of the stack are in continuous change.

We also investigated the causes of performance degradation in distributed file systems, focusing on the tail latency of read operations. First, we characterized the performance of a distributed file system (*i.e.*, HDFS) in Microsoft's resource harvesting datacenters. We showed that disk I/O contention is the primary cause of tail latency (in our datacenters). We then proposed smart hedging, a technique to mitigate tail latency in read operations. Smart hedging reacts to tail latency, no matter the root cause of the variation. Moreover, our technique is production compliant, it effectively deals with heterogeneity (static and dynamic), massive scale, resource harvesting, load imbalance, and maintainability and ease of deployment. Our technique reduces tail latencies by tracking latencies on the client side and establishes a dynamic threshold. A hedge (*i.e.*, duplicate request) is triggered when a request surpasses the dynamic latency threshold in an exponential back off manner. We implemented our solution in HDFS, as part of a client library. We evaluated smart hedging with synthetic workloads in a testbed with 4K servers. We also evaluated our technique with production workloads on 70K servers across 3 Microsoft's datacenters. We compared our solution against baseline HDFS and a production version of HDFS with regular hedging. We showed that it is possible to create techniques to reduce read operations tail latency in distributed file systems under production constraints. Our technique is deployed in Microsoft's resource-harvesting datacenters.

References

[1] Adaptive file readahead. https://lwn.net/Articles/155510/.

[2] Amazon Web Services. https://aws.amazon.com.

[3] Blktrace. http://git.kernel.dk/cgit/blktrace/.

[4] Cisco Global Cloud Index: Forecast and Methodology, 2016–2021 White Paper. https://www.cisco.com/c/en/us/solutions/collateral/service-provider/global-cloud-index-gci/white-paper-c11-738085.html.

[5] Fio - Flexible I/O Tester. https://fio.readthedocs.io/en/latest/fio_doc.html.

[6] Google Cloud Platform. https://cloud.google.com.

[7] Hadoop filesystem at Twitter. https://blog.twitter.com/engineering/en_us/a/2015/hadoop-filesystem-at-twitter.html.

[8] Microsoft Azure Cloud Computing. https://azure.microsoft.com.

[9] Nutanix Hardware Platforms. https://www.nutanix.com/products/hardware-platforms/.

[10] Perf. https://github.com/torvalds/linux/tree/master/tools/perf/Documentation.

[11] Strace. https://strace.io/.

[12] HDFS Architecture Guide, 2008. https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html.

[13] MongoDB Managed Chain Replication, 2008. https://docs.mongodb.com/manual/tutorial/manage-chained-replication/.

[14] Apache HTrace: A tracing framework for use with distributed systems, 2017. http://htrace.incubator.apache.org/.

[15] DistCp Guide, 2017. http://hadoop.apache.org/docs/current/hadoop-distcp/DistCp.html.

[16] TeraGen, 2017. https://hadoop.apache.org/docs/current/api/org/apache/hadoop/examples/teragen/package-summary.html.

[17] Track time to process packet in Datanode, 2017. https://issues.apache.org/jira/browse/HDFS-13053.

[18] Track speed in DFSClient, 2018. https://issues.apache.org/jira/browse/HDFS-12861.

[19] AGHAYEV, A., TS'O, T., GIBSON, G., AND DESNOYERS, P. Evolving Ext4 for Shingled Disks. In *15th USENIX Conference on File and Storage Technologies (FAST 17)* (Santa Clara, CA, 2017), USENIX Association.

[20] AMVROSIADIS, G., BUTT, A. R., TARASOV, V., ZADOK, E., ZHAO, M., AHMAD, I., ARPACI-DUSSEAU, R. H., CHEN, F., CHEN, Y., CHEN, Y., CHENG, Y., CHIDAMBARAM, V., DA SILVA, D., DEMKE-BROWN, A., DESNOYERS, P., FLINN, J., HE, X., JIANG, S., KUENNING, G., LI, M., MALTZAHN, C., MILLER, E. L., MOHROR, K., RANGASWAMI, R., REDDY, N., ROSENTHAL, D., TOSUN, A. S., TALAGALA, N., VARMAN, P., VAZHKUDAI, S., WALDANI, A., ZHANG, X., ZHANG, Y., AND ZHENG, M. Data Storage Research Vision 2025: Report on NSF Visioning Workshop Held May 30–June 1, 2018. Tech. rep., USA, 2018.

[21] AMVROSIADIS, G., PARK, J. W., GANGER, G. R., GIBSON, G. A., BASEMAN, E., AND DEBARDELEBEN, N. On the diversity of cluster workloads and its impact on research results. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)* (Boston, MA, 2018), USENIX Association.

[22] ANANTHANARAYANAN, G., GHODSI, A., SHENKER, S., AND STOICA, I. Effective Straggler Mitigation: Attack of the Clones. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)* (Lombard, IL, 2013), USENIX Association.

[23] ANANTHANARAYANAN, G., KANDULA, S., GREENBERG, A., STOICA, I., LU, Y., SAHA, B., AND HARRIS, E. Reining in the Outliers in Map-reduce Clusters Using Mantri. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI 10)* (Vancouver, BC, Canada, 2010), USENIX Association.

[24] ANDERSEN, D. G., FRANKLIN, J., KAMINSKY, M., PHANISHAYEE, A., TAN, L., AND VASUDEVAN, V. Fawn: A fast array of wimpy nodes. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles (SOSP 09)* (Big Sky, Montana, USA, 2009), ACM.

[25] APACHE SOFTWARE FOUNDATION. Apache Hadoop.

[26] APACHE SOFTWARE FOUNDATION. Apache HBase.

[27] APACHE SOFTWARE FOUNDATION. Apache Spark.

[28] BALAKRISHNAN, M., MALKHI, D., PRABHAKARAN, V., WOBBLER, T., WEI, M., AND DAVIS, J. D. CORFU: A Shared Log Design for Flash Clusters. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)* (San Jose, CA, 2012), USENIX Association.

[29] BORGE, M. F., DINU, F., AND ZWAENEPOEL, W. Understanding and Taming SSD Read Performance Variability: HDFS case study. *arXiv preprint arXiv:1903.09347* (2019).

[30] BRUTLAG, J. Speed Matters. https://ai.googleblog.com/2009/06/speed-matters.html.

[31] CAI, Y., HARATSCH, E. F., MUTLU, O., AND MAI, K. Error Patterns in MLC NAND Flash Memory: Measurement, Characterization, and Analysis. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE 12)* (Dresden, Germany, 2012), EDA Consortium.

[32] CAI, Y., LUO, Y., GHOSE, S., AND MUTLU, O. Read Disturb Errors in MLC NAND Flash Memory: Characterization, Mitigation, and Recovery. In *Proceedings of the 2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 15)* (Rio de Janeiro, Brazil, 2015), IEEE Computer Society.

[33] CAI, Y., LUO, Y., HARATSCH, E. F., MAI, K., AND MUTLU, O. Data Retention in MLC NAND Flash Memory: Characterization, Optimization, and Recovery. In *21st IEEE International Symposium on High Performance Computer Architecture (HPCA 2015)* (Burlingame, CA, USA, 2015), IEEE Computer Society.

[34] CALDER, B., WANG, J., OGUS, A., NILAKANTAN, N., SKJOLSVOLD, A., MCKELVIE, S., XU, Y., SRIVASTAV, S., WU, J., SIMITCI, H., HARIDAS, J., UDDARAJU, C., KHATRI, H., EDWARDS, A., BEDEKAR, V., MAINALI, S., ABBASI, R., AGARWAL, A., HAQ, M. F. U., HAQ, M. I. U., BHARDWAJ, D., DAYANAND, S., ADUSUMILLI, A., MCNETT, M., SANKARAN, S., MANIVANNAN, K., AND RIGAS, L. Windows Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP 11)* (Cascais, Portugal, 2011), ACM.

[35] CANO, I., AIYAR, S., AND KRISHNAMURTHY, A. Characterizing Private Clouds: A Large-Scale Empirical Analysis of Enterprise Clusters. In *Proceedings of the Seventh ACM Symposium on Cloud Computing (SoCC 16)* (Santa Clara, CA, USA, 2016), ACM.

[36] CAO, Z., TARASOV, V., RAMAN, H. P., HILDEBRAND, D., AND ZADOK, E. On the Performance Variation in Modern Storage Stacks. In *15th USENIX Conference on File and Storage Technologies (FAST 17)* (Santa Clara, CA, USA, 2017), USENIX Association.

[37] CARPENTER, J., AND HEWITT, E. *Cassandra: The Definitive Guide: Distributed Data at Web Scale*. " O'Reilly Media, Inc.", 2016.

[38] CHAIKEN, R., JENKINS, B., LARSON, P.-Å., RAMSEY, B., SHAKIB, D., WEAVER, S., AND ZHOU, J. SCOPE: Easy and Efficient Parallel Processing of Massive Data Sets. *Proceedings of the VLDB Endowment (VLDB 08)* (2008).

[39] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. E. Bigtable: A Distributed Storage System for Structured Data. *ACM Transactions on Computer Systems (TOCS)* (2008).

[40] CONWAY, A., BAKSHI, A., JIAO, Y., JANNEN, W., ZHAN, Y., YUAN, J., BENDER, M. A., JOHNSON, R., KUSZMAUL, B. C., PORTER, D. E., AND FARACH-COLTON, M. File Systems Fated for Senescence? Nonsense, Says Science! In *15th USENIX Conference on File and Storage Technologies (FAST 17)* (Santa Clara, CA, USA, 2017), USENIX Association.

[41] DEAN, J., AND BARROSO, L. A. The tail at scale. *Communications of the ACM* (2013).

[42] DEAN, J., AND GHEMAWAT, S. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation (OSDI 04)* (San Francisco, CA, USA, 2004), USENIX Association.

[43] DELIMITROU, C., AND KOZYRAKIS, C. Quasar: Resource-efficient and QoS-aware Cluster Management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 14)* (Salt Lake City, Utah, USA, 2014), ACM.

[44] DINU, F., AND NG, T. Understanding the Effects and Implications of Compute Node Related Failures in Hadoop. In *Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing (HPDC 12)* (Delft, The Netherlands, 2012), ACM.

[45] ESCRIVA, R., WONG, B., AND SIRER, E. G. HyperDex: A Distributed, Searchable Key-value Store.

[46] FACEBOOK OPEN SOURCE. RocksDB. https://rocksdb.org/.

[47] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. The Google File System. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (SOSP 03)* (Bolton Landing, NY, USA), ACM.

[48] GODER, A., SPIRIDONOV, A., AND WANG, Y. Bistro: Scheduling Data-Parallel Jobs Against Live Production Systems. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)* (Santa Clara, CA, USA, 2015), USENIX Association.

[49] GULATI, A., AHMAD, I., AND WALDSPURGER, C. A. PARDA: Proportional allocation of resources for distributed storage access. In *7th USENIX Conference on File and Storage Technologies (FAST 09)* (San Francisco, CA, USA, 2009), USENIX Association.

[50] GUNAWI, H. S., SUMINTO, R. O., SEARS, R., GOLLIHER, C., SUNDARARAMAN, S., LIN, X., EMAMI, T., SHENG, W., BIDOKHTI, N., MCCAFFREY, C., GRIDER, G., FIELDS, P. M., HARMS, K., ROSS, R. B., JACOBSON, A., RICCI, R., WEBB, K., ALVARO, P., RUNESHA, H. B., HAO, M., AND LI, H. Fail-Slow at Scale: Evidence of Hardware Performance Faults in Large Production Systems. In *16th*

USENIX Conference on File and Storage Technologies (FAST 18) (Oakland, CA, USA, 2018), USENIX Association.

[51] HAO, M., LI, H., TONG, M. H., PAKHA, C., SUMINTO, R. O., STUARDO, C. A., CHIEN, A. A., AND GUNAWI, H. S. Mittos: Supporting millisecond tail tolerance with fast rejecting slo-aware os interface. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP 17)* (Shanghai, China, 2017), ACM.

[52] HAO, M., SOUNDARARAJAN, G., KENCHAMMANA-HOSEKOTE, D. R., CHIEN, A. A., AND GUNAWI, H. S. The Tail at Store: A Revelation from Millions of Hours of Disk and SSD Deployments. In *14th USENIX Conference on File and Storage Technologies (FAST 16)* (Santa Clara, CA, USA, 2016), USENIX Association.

[53] HARTER, T., BORTHAKUR, D., DONG, S., AIYER, A. S., TANG, L., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Analysis of HDFS Under HBase: A Facebook Messages Case Study. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST 14)* (Santa Clara, CA, USA, 2014), USENIX Association.

[54] HE, J., KANNAN, S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. The Unwritten Contract of Solid State Drives. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys 17)* (Belgrade, Serbia, 2017), ACM.

[55] HE, J., NGUYEN, D., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Reducing File System Tail Latencies with Chopper. In *13th USENIX Conference on File and Storage Technologies (FAST 15)* (Santa Clara, CA, USA, 2015), USENIX Association.

[56] IBM. 10 Key Marketing Trends for 2017.

[57] IORGULESCU, C., AZIMI, R., KWON, Y., ELNIKETY, S., SYAMALA, M., NARASAYYA, V., HERODOTOU, H., TOMITA, P., CHEN, A., ZHANG, J., AND WANG, J. PerfIso: Performance Isolation for Commercial Latency-Sensitive Services. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)* (Boston, MA, USA, 2018), USENIX Association.

[58] JALAPARTI, V., BODIK, P., KANDULA, S., MENACHE, I., RYBALKIN, M., AND YAN, C. Speeding up Distributed Request-Response Workflows. In *Proceedings of the ACM SIGCOMM 2013 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM 13)* (Hong Kong, China, 2013), ACM.

[59] JANNEN, W., YUAN, J., ZHAN, Y., AKSHINTALA, A., ESMET, J., JIAO, Y., MITTAL, A., PANDEY, P., REDDY, P., WALSH, L., ET AL. BetrFS: A Right-Optimized Write-Optimized File System. In *13th USENIX Conference on File and Storage Technologies (FAST 15)* (Santa Clara, CA, USA, 2015), USENIX Association.

[60] JIN, W., CHASE, J. S., AND KAUR, J. Interposed Proportional Sharing for a Storage Service Utility. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 04)* (New York, NY, USA, 2004), ACM.

[61] JUNG, M., AND KANDEMIR, M. Revisiting Widely Held SSD Expectations and Rethinking System-level Implications. In *Proceedings of the ACM SIGMETRICS/International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 13)* (Pittsburgh, PA, USA, 2013), ACM.

[62] JYOTHI, S. A., CURINO, C., MENACHE, I., NARAYANAMURTHY, S. M., TUMANOV, A., YANIV, J., MAVLYUTOV, R., GOIRI, Í., KRISHNAN, S., KULKARNI, J., AND RAO, S. Morpheus: Towards Automated SLOs for Enterprise Clusters. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (Savannah, GA, USA, 2016), USENIX Association.

[63] KADEKODI, S., NAGARAJAN, V., GANGER, G. R., AND GIBSON, G. A. Geriatrix: Aging what you see and what you don't see. a file system aging approach for modern storage systems. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference* (2018), USENIX Association, pp. 691–703.

[64] KWON, Y., FINGLER, H., HUNT, T., PETER, S., WITCHEL, E., AND ANDERSON, T. Strata: A Cross Media File System. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP 17)* (Shanghai, China, 2017), ACM.

[65] LAKSHMAN, A., AND MALIK, P. Cassandra: a Decentralized Structured Storage System. ACM.

[66] LEE, C., SIM, D., HWANG, J. Y., AND CHO, S. F2FS: A New File System for Flash Storage. In *13th USENIX Conference on File and Storage Technologies (FAST 15)* (Santa Clara, CA, USA, 2015), USENIX Association.

[67] LEE, E., HAN, Y., YANG, S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. How to Teach an Old File System Dog New Object Store Tricks. In *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18)* (Boston, MA, USA, 2018), USENIX Association.

[68] LI, H., GHODSI, A., ZAHARIA, M., SHENKER, S., AND STOICA, I. Tachyon: Reliable, Memory Speed Storage for Cluster Computing Frameworks. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC 14)* (Seattle, WA, USA, 2014), ACM.

[69] LI, J., SHARMA, N. K., PORTS, D. R. K., AND GRIBBLE, S. D. Tales of the Tail: Hardware, OS, and Application-level Sources of Tail Latency. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC 14)* (Seattle, WA, USA, 2014), ACM.

[70] LO, D., CHENG, L., GOVINDARAJU, R., RANGANATHAN, P., AND KOZYRAKIS, C. Heracles: Improving Resource Efficiency at Scale. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture (ISCA 15)* (Portland, OR, USA, 2015), ACM.

[71] MACE, J., BODIK, P., FONSECA, R., AND MUSUVATHI, M. Retro: Targeted Resource Management in Multi-tenant Distributed Systems. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)* (Oakland, CA, USA, 2015), USENIX Association.

[72] MARICQ, A., DUPLYAKIN, D., JIMENEZ, I., MALTZAHN, C., STUTSMAN, R., AND RICCI, R. Taming Performance Variability. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)* (Carlsbad, CA, USA, 2018), USENIX Association.

[73] MARS, J., TANG, L., HUNDT, R., SKADRON, K., AND SOFFA, M. L. Bubble-Up: Increasing Utilization in Modern Warehouse Scale Computers via Sensible Co-locations. In *Proceedings of the 44th annual IEEE/ACM International Symposium on Microarchitecture (MICRO 11)* (Porto Alegre, Brazil, 2011), ACM.

[74] MATHUR, A., CAO, M., BHATTACHARYA, S., DILGER, A., TOMAS, A., AND VIVIER, L. The new ext4 filesystem: current status and future plans. In *Proceedings of the Linux Symposium* (Ottawa, Canada, 2007).

[75] MEZA, J., WU, Q., KUMAR, S., AND MUTLU, O. A large-scale study of flash memory failures in the field. In *Proceedings of the 41st ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '15)* (Portland, Oregon, USA, 2015), ACM.

[76] MISRA, P., GOIRI, I., KACE, J., AND BIANCHINI, R. Scaling Distributed File Systems in Resource-Harvesting Datacenters. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)* (Santa Clara, CA, USA, 2017), USENIX Association.

[77] MISRA, P. A., BORGE, M. F., GOIRI, Í., LEBECK, A. R., ZWAENEPOEL, W., AND BIANCHINI, R. Managing Tail Latency in Datacenter-Scale File Systems Under Production Constraints. In *Proceedings of the Fourteenth European Conference on Computer Systems (EuroSys 19)* (Dresden, Germany, 2019), ACM.

[78] NIAZI, S., ISMAIL, M., HARIDI, S., DOWLING, J., GROHSSCHMIEDT, S., AND RONSTRÖM, M. HopsFS: Scaling Hierarchical File System Metadata Using NewSQL Databases. In *15th USENIX Conference on File and Storage Technologies (FAST 17)* (Santa Clara, CA, USA, 2017), USENIX Association.

[79] OUYANG, X., GARRAGHAN, P., PRIMAS, B., MCKEE, D., TOWNEND, P., AND XU, J. Adaptive speculation for efficient internetware application execution in clouds. *ACM Transactions on Internet Technology (TOIT)* (2018).

[80] PAI, R., PULAVARTY, B., AND CAO, M. Linux 2.6 performance improvement through readahead optimization. In *Proceedings of the Linux Symposium* (2004), vol. 2, pp. 105–116.

[81] REDA, W., CANINI, M., SURESH, L., KOSTIĆ, D., AND BRAITHWAITE, S. Rein: Taming Tail Latency in Key-Value Stores via Multiget Scheduling. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys 17)* (Belgrade, Serbia, 2017), ACM.

[82] REISS, C., TUMANOV, A., GANGER, G. R., KATZ, R. H., AND KOZUCH, M. A. Heterogeneity and Dynamicity of Clouds at Scale: Google Trace Analysis. In *Proceedings of the Third ACM Symposium on Cloud Computing (SoCC 12)* (San Jose, CA, USA, 2012), ACM.

[83] REN, X., ANANTHANARAYANAN, G., WIERMAN, A., AND YU, M. Hopper: Decentralized Speculation-aware Cluster Scheduling at Scale. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication (SIGCOMM 15)* (London, United Kingdom, 2015), ACM.

[84] SCHROEDER, B., AND GIBSON, G. A Large-scale Study of Failures in High-performance Computing Systems. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN 06)* (Washington, DC, USA, 2006), IEEE Computer Society.

[85] SEAGATE, AND IDC. Data Age 2025: The Evolution of Data to Life-Critical Don't Focus on Big Data; Focus on the Data That's Big.

[86] SEAGATE, AND IDC. The Digitization of the World.

[87] SHAFER, J., RIXNER, S., AND COX, A. L. The Hadoop Distributed Filesystem: Balancing Portability and Performance. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS 2010)* (White Plains, NY, USA, 2010), IEEE Computer Society.

[88] SHUE, D., FREEDMAN, M. J., AND SHAIKH, A. Performance Isolation and Fairness for Multi-tenant Cloud Storage. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)* (Hollywood, CA, USA, 2012), USENIX Association.

[89] SHVACHKO, K., KUANG, H., RADIA, S., AND CHANSLER, R. The Hadoop Distributed File System. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST 10)* (Incline Village, NV, USA, 2010), IEEE Computer Society.

[90] SKOURTIS, D., ACHLIOPTAS, D., WATKINS, N., MALTZAHN, C., AND BRANDT, S. A. Flash on Rails: Consistent Flash Performance through Redundancy. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)* (Philadelphia, PA, USA, 2014), USENIX Association.

[91] SOUDERS, S. Velocity and the Bottom Line.

[92] SUMINTO, R. O., STUARDO, C. A., CLARK, A., KE, H., LEESATAPORNWONGSA, T., FU, B., KURNIAWAN, D. H., MARTIN, V., UMA, M. R. G., AND GUNAWI, H. S. PBSE: A Robust Path-based Speculative Execution for Degraded-network Tail Tolerance in Data-parallel Frameworks. In *Proceedings of the 2017 Symposium on Cloud Computing (SoCC 17)* (Santa Clara, CA, USA, 2017), ACM.

[93] SURESH, P. L., CANINI, M., SCHMID, S., AND FELDMANN, A. C3: Cutting Tail Latency in Cloud Data Stores via Adaptive Replica Selection. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)* (Oakland, CA, USA, 2015), USENIX Association.

[94] SWEENEY, A., DOUCETTE, D., HU, W., ANDERSON, C., NISHIMOTO, M., AND PECK, G. Scalability in the XFS File System. In *Proceedings of the 1996 Annual Conference on USENIX Annual Technical Conference (USENIX ATC 96)* (San Diego, CA, USA, 1996), USENIX Association.

[95] TANG, S., LEE, B., AND HE, B. DynamicMR: A Dynamic Slot Allocation Optimization Framework for MapReduce Clusters. IEEE Computer Society.

[96] TERRACE, J., AND FREEDMAN, M. J. Object Storage on CRAQ: High-throughput Chain Replication for Read-mostly Workloads. In *Proceedings of the 2009 Conference on USENIX Annual Technical Conference (USENIX ATC 09)* (San Diego, CA, USA, 2009), USENIX Association.

[97] TRUSHKOWSKY, B., BODÍK, P., FOX, A., FRANKLIN, M. J., JORDAN, M. I., AND PATTERSON, D. A. The SCADS Director: Scaling a Distributed Storage System Under Stringent Performance Requirements. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies (FAST 11)* (San Jose, California, 2011), USENIX Association.

[98] VAN RENESSE, R., AND SCHNEIDER, F. B. "chain replication for supporting high throughput and availability". In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation (OSDI 04)* (San Francisco, CA, 2004), USENIX Association.

[99] VANGOOR, B. K. R., TARASOV, V., AND ZADOK, E. "to fuse or not to fuse: Performance of user-space file systems."". In *15th USENIX Conference on File and Storage Technologies (FAST 17)* (Santa Clara, CA, 2017), USENIX Association.

[100] VEERARAGHAVAN, K., MEZA, J., CHOU, D., KIM, W., MARGULIS, S., MICHELSON, S., NISHTALA, R., OBENSHAIN, D., PERELMAN, D., AND SONG, Y. J. Kraken: Leveraging Live Traffic Tests to Identify and Resolve Resource Utilization Bottlenecks in Large Scale Web Services. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (Savannah, GA, 2016), USENIX Association.

[101] VERMA, A., PEDROSA, L., KORUPOLU, M., OPPENHEIMER, D., TUNE, E., AND WILKES, J. Large-Scale Cluster Management at Google with Borg. In *Proceedings of the Tenth European Conference on Computer Systems (EuroSys 15)* (Bordeaux, France, 2015), ACM.

[102] WANG, A., VENKATARAMAN, S., ALSPAUGH, S., KATZ, R., AND STOICA, I. Cake: enabling high-level SLOs on shared storage systems. In *Proceedings of the Third ACM Symposium on Cloud Computing (SoCC 12)* (San Jose, California, 2012), ACM.

[103] WU, F., XI, H., LI, J., AND ZOU, N. Linux readahead: less tricks for more. In *Proceedings of the Linux Symposium* (2007), vol. 2, Citeseer, pp. 273–284.

[104] WU, Z., BUTKIEWICZ, M., PERKINS, D., KATZ-BASSETT, E., AND MADHYASTHA, H. V. SPANStore: Cost-effective geo-replicated storage spanning multiple cloud services. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP 13)* (Farminton, Pennsylvania, 2013), ACM.

[105] WU, Z., YU, C., AND MADHYASTHA, H. V. CosTLO: Cost-Effective Redundancy for Lower Latency Variance on Cloud Storage Services. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)* (Oakland, CA, 2015), USENIX Association.

[106] XE, L. Support Hedged Reads in DFSClient, 2014. https://issues.apache.org/jira/browse/HDFS-5776.

[107] XE, L., AND MCCABE, C. P. Support Non-Positional Hedged Reads in HDFS, 2017. https://issues.apache.org/jira/browse/HDFS-6450.

[108] YAN, S., LI, H., HAO, M., TONG, M. H., SUNDARARAMAN, S., CHIEN, A. A., AND GUNAWI, H. S. Tiny-Tail Flash: Near-Perfect Elimination of Garbage Collection Tail Latencies in NAND SSDs. In *15th USENIX Conference on File and Storage Technologies (FAST 17)* (Santa Clara, CA, 2017), USENIX Association.

[109] YANG, H., BRESLOW, A., MARS, J., AND TANG, L. Bubble-flux: Precise Online QoS Management for Increased Utilization in Warehouse Scale Computers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA 13)* (Tel-Aviv, Israel, 2013), ACM.

[110] YANG, S., HARTER, T., AGRAWAL, N., KOWSALYA, S. S., KRISHNAMURTHY, A., AL-KISWANY, S., KAUSHIK, R. T., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Split-level I/O Scheduling. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP 15)* (Monterey, CA, USA, 2015), ACM.

[111] YANG, X., BLACKBURN, S. M., AND MCKINLEY, K. S. Elfen scheduling: Fine-grain principled borrowing from latency-critical workloads using simultaneous multithreading. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)* (Denver, CO, USA, 2016), Usenix Association.

[112] YANG, Y., KIM, G.-W., SONG, W. W., LEE, Y., CHUNG, A., QIAN, Z., CHO, B., AND CHUN, B.-G. Pado: A Data Processing Engine for Harnessing Transient Resources in Datacenters. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys 17)* (Belgrade, Serbia, 2017), ACM.

[113] ZAHARIA, M., KONWINSKI, A., JOSEPH, A. D., KATZ, R., AND STOICA, I. Improving mapreduce performance in heterogeneous environments. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI 08)* (San Diego, CA, USA, 2008), USENIX Association.

[114] ZHANG, J., SIVASUBRAMANIAM, A., WANG, Q., RISKA, A., AND RIEDEL, E. Storage Performance Virtualization via Throughput and Latency Control. In *Transactions On Storage* (2006).

[115] ZHANG, Y., PREKAS, G., FUMAROLA, G. M., FONTOURA, M., GOIRI, I., AND BIANCHINI, R. History-Based Harvesting of Spare Cycles and Storage in Large-Scale Datacenters. In *12th USENIX Symposium on Operating Systems Design and Implementation ((OSDI) 16)* (Savannah, GA, USA, 2016), USENIX Association.

[116] ZHU, T., TUMANOV, A., KOZUCH, M. A., HARCHOL-BALTER, M., AND GANGER, G. R. PriorityMeister: Tail Latency QoS for Shared Networked Storage. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC 14)* (Seattle, WA, USA, 2014), ACM.