Storage System Tracing and Analysis

by

Dutch T. Meyer

B.Sc., The University of Washington, 2001
B.A., The University of Washington, 2001
M.Sc., The University of British Columbia, 2008

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

in

The Faculty of Graduate and Postdoctoral Studies

(Computer Science)

THE UNIVERSITY OF BRITISH COLUMBIA

(Vancouver)

August 2015

Abstract

File system studies are critical to the accurate configuration, design, and continued evolution of storage systems. However, surprisingly little has been published about the use and behaviour of file systems under live deployment. Most published file system studies are years old, or are extremely limited in size and scope. One reason for this deficiency is that studying clusters of storage systems operating at scale requires large data repositories that are difficult to collect, manage, and query. In effect, the demands introduced by performing detailed file system traces creates new and interesting storage challenges in and of itself. This thesis describes a methodology to facilitate the collection, analysis, and publication of large traces of file system activity and structure so that organizations can perform regular analysis of their storage systems and retain that analysis to answer questions about their system's behaviour. To validate this methodology I investigate the use and performance of several large storage deployments. I consider the impact of the observed system usage and behaviour on file system design, and I describe the processes by which the collected traces can be efficiently processed and manipulated. I report on several examples of long standing incorrect assumptions, efficient engineering alternatives, and new opportunities in storage system design.

Preface

Except where specifically stated otherwise, the contents of this thesis are my own. The research draws primarily from two published research papers which I have extended here with further unpublished analysis into the datasets gathered in the original work.

Versions of figures 5.1, 5.2, 5.3, 5.4, 5.5, 5.7, 5.8, 5.9, 5.11, 7.1, 7.2, 7.3, 7.4, 7.5, 7.6, and 7.8 as well as Tables 5.1, and 5.2 appeared in "A Study of Practical Deduplication" published by USENIX in 2010, in which I was the lead investigator [MB11]. The research was conducted at Microsoft Corporation in Redmond, Washington. After the initial draft of this paper, which I wrote, Bill Bolosky and I collaborated in the authorship of subsequent drafts and the final research paper. Modified portions of this collaborative text are used with permission in Chapters 5, 6, and 7. Bill Bolosky wrote the SQL functions in Appendix A along with other unlisted routines used to generate the processed data from which the graphs which reference this dataset are generated.

Figure 6.1 was drawn by my advisor Andrew Warfield and myself as part of "Capo: Recapitulating Storage for Virtual Desktops" published by USENIX in 2010 [SMW+10]. Other co-authors of the Capo paper include Mohammad Shamma, Jake Wires, Maria Ivanova, and Norm Hutchinson.

The excerpts of the paper that appear here, including Figures 6.2, 6.3, 6.4, 6.5, 6.6, 6.7, 7.9, 7.10, 7.11, Table 7.1, and the original text that I modified for this thesis in Chapters 6 and 7 are all my own contributions to that paper. I conducted the research and gathering the data presented in these chapters. Maria Ivanova helped write some of the tools used to process the data in Figures 6.6 and 6.4.

Table of Contents

Appendices

List of Tables

List of Figures

Acknowledgements

I would like to thank my advisor Andrew Warfield for his continuous guidance and support thoughout my Ph.D. and for so much helpful collaboration in the research and publishing process. I have taken every opportunity to ruthlessly steal as many lessons from him as possible, and I have no intention of stopping any time soon. I would also like to thank my collaborators at UBC, Mohammad Shamma, Jake Wires, and Maria Ivanova with whom I coauthored one paper which contibuted to this thesis. Each contributed to the design and evaluation of the Capo system, which is referenced here. I would also thank Geoffrey Lefebvre and Brandan Cully for their guidance as senior graduate students throughout my research.

I would like to thank Microsoft for enabling and supporting analytical research, and ultimately for allowing the results to be released, complete with a public data repository. Bill Bolosky was critical in guiding me through the mechanics of conducting the disk content study and Richard Draves assisted with the bureaucracy. Bill Bolosky wrote most of the sql queries that were used to collect the data from disk content study database after my internship had ended. The University of British Columbia IT department was also very generous in facilitating data collection and installing the tracing software, particularly Brent Dunnington.

Dedication

To Nanners and Finnegan and Luna. I had no idea when I started this work how much effort each of you would put into my completing it. It is no exaggeration to say that your support and sacrifices made this possible.

Chapter 1

Introduction

The performance and efficiency of enterprise storage has grown in importance as increased CPU and memory performance, combined with high bandwidth networks and dense virtualization, have exacerbated the storage bottleneck. At the same time, storage systems are growing more feature rich and thus more complicated. Unfortunately, file systems and the tools that support them have changed relatively little in their ability to elucidate the workload and data organization that define the storage system's behaviour. We frequently operate with surprisingly little specific knowledge describing what our storage systems are doing.

The lack of insight into storage system behaviour leads users and designers to accept inefficiencies that we would not otherwise tolerate. System architects miss opportunities to improve their systems because those opportunities are hidden in the system's workload. For example, one recent investigation into a specific data center workload found that a typical piece of data in a widely used storage architecture is relocated on disk 17 times when being written [HBD+14] - such a problem is relatively easy to address, once identified. Related problems hamper storage system administrators who have very little analytic data upon which to base the decision to deploy

optional features. For example, prior to the work in this thesis, there was no publicly available measurement of the compression provided by deduplication on general purpose datasets of any significant size. This leaves administrators without any basis to select from various deduplication options aside from their own direct experience. Finally, workloads initiated by users are sometimes wholly unnecessary, such as defragmentation applied by default over a shared file system, where locality may have little impact. Such a condition is particularly likely to remain unnoticed in dense computing environments where user workloads are hidden behind several layers of virtualization.

Traditionally, analysis of storage system behaviour has been difficult, tedious, and costly, which too often leads it to be extremely limited in scope and kept private. To address these challenges, I present several advances in, and a collection of findings from, the measurement of data storage systems. I draw my contributions from two case studies of live system tracing and analysis, in two different large commercially deployed enterprise storage environments. Although there are relatively few similar studies available in the published literature, this work is additionally noteworthy for considering detailed traces of large numbers of systems over long time scales, and in environments that are historically difficult to collect data from. I present an analysis of the data obtained in these case studies, which is useful for both designers and researchers. These insights include observations drawn from features that are not typically considered in depth, such as the effects of virtualization and deduplication. I also present the data analysis techniques that have made obtaining these results possible, which are themselves novel

and useful to researchers wishing to recreate my work and apply it to other studies. In addition, I present the data collection techniques and tools used to gather this data, which facilitate highly detailed trace collection with low impact for those who would apply or adapt my methodology to other systems. Finally, to enable other researchers to conduct further analysis against these datasets, I have shared and published most of the data collected in this work in the rawest forms possible, which has required novel approaches to data anonymization. These contributions constitute a framework for organizations to frequently gather, persist, analyze, and share detailed storage traces that are meant to be sufficiently simple that it can be be deployed by enterprise system administrators.

1.1 Thesis Statement

The thesis of my work is that *the growing complexity of modern storage systems can be better met than it is presently, by organizations periodically collecting and retaining large, extremely detailed traces of system state and behaviour.*

I define growing complexity as an increase in I/O intensive workloads, larger datasets, and more feature-rich storage systems. The trace-oriented approach that I describe here can be applied quite broadly to address storage system complexity by providing insight into many aspects of storage system operation. To support this thesis I focus on three applications of tracing and analysis: Capacity Management, Workload Introspection, and Namespace Modelling.

Specifically, I claim:

- **Capacity Management:** That detailed tracing and analysis can be applied to manage disk capacity more effectively than is commonly done today.
- **Workload Introspection:** First, that there are significant areas of misplaced effort in the operation of storage systems for enterprise workstations. And second, that tracing and analysis provides a mechanism for eliminating that wasted effort by highlighting the otherwise hidden behaviour of storage workloads.
- **Namespace Modelling:** That detailed analysis of the structure of file system namespaces, although rarely done, can inform and justify new storage features. In addition, that existing storage systems can directly exploit knowledge of file system namespace organization to improve performance.

In addition, throughout this thesis, based on the case studies I have performed, I present a significant and broad body of new insights about how current systems operate in the field, and how they might be improved.

1.2 Thesis Organization

In Chapter 2 I briefly expand on the complexity of modern storage systems by describing the structure of Microsoft Azure Storage, a large cloud-based enterprise storage system.

I then detail the background and motivation that drives the study of storage systems in Chapter 3. I discuss the differences between live system tracing and studying static on-disk data, and describe some of the design choices available in performing studies of each type. I then describe the case studies used to validate this thesis in the context of these choices and discuss related work.

In Chapter 4 I describe the tools I have developed for data collection and analysis, including the process used to install, collect, analyze and anonymize traces. I also discuss the challenges and costs of gathering and retaining traces and the lessons drawn from the process of analyzing the data, limitations and biases in the dataset, and possible attacks to the anonymization process.

In Chapter 5 I present a case study addressing **Capacity Management** in enterprise storage. In this chapter I study a large number of file systems to characterize recent changes to disk size and utilization, and measure the effectiveness of a widely deployed but poorly understood feature - deduplication.

In Chapter 6, I present a set of case studies addressing **Workload Introspection** in virtualized storage environments. I first provide the background information necessary to understand the virtualized environment and discuss some of the administrative challenges it poses. I then demonstrate that collecting traces of individual storage requests at much higher detail than is typical can support novel actionable investigations into system behaviour. I specifically show how tracing can identify wasted efforts in user workloads and how to eliminate them, and how analysis can measure the effectiveness

of scheduled background defragmentation, which is a workload-oriented concern that administrators have long shared.

In Chapter 7, I present a case study investigating the topology of file system namespaces and the distribution of files within them, which I refer to as **Namespace Modelling**. I apply this to enterprise desktop workloads in two ways. I first compare the namespace models I have gathered to otherwise unchecked assertions from prior research efforts. I then go on to show that enterprise workloads can benefit from leveraging detailed traces of file system namespaces by illustrating new opportunities for optimization.

I conclude in Chapter 8 that there are many examples of simple improvements to existing deployed storage systems that are made obvious by tracing and analysis. This finding lends support to the hypothesis that there is benefit to organizations performing detailed traces of file system studies and workload.

Chapter 2

A Brief Case Study: Storage System Complexity

Today, enterprise storage systems operate in a broad range of environments and exist at many different scales. However, all but the smallest systems face similar fundamental challenges: They must serve a large number of I/O operations per second, they must store a large amount of data, and they are pressured to expose complex interfaces and feature sets. The scale of each of these requirements is significantly different from storage workloads a decade ago, and it drives these system to be very complex.

In this chapter, to help contextualize this increase in storage system complexity, I present a brief description of the structure of one enterprise storage system. Except where otherwise stated, I draw the entirety of this simplified [1] example from the description provided in 2011 in "Windows Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency" [CWO+11].

Microsoft Azure Storage is a cloud service in widespread use since 2010,

[1] In this example I omit many aspects of Azure's operation and have made changes to some component names, for the purpose of explaining the relevant structural aspects of the system succinctly.

which provides customers with pay-for-use online storage. Customers are free to grow their usage without explicit limits and the underlying architecture scalably accommodates the load. Its interface natively supports files (called blobs), tables of structure data, and queues for message delivery. Each of these objects is native to the file system and is accessed with an independent API.

A diagram of the Azure Storage architecture is provided in Figure 2.1. Like most storage systems, it is structured in a stack of layers, each with a well defined narrow interface. The top of the stack is the layer responsible for load balancing. The load balancing layer passes requests to the partition layer, which is responsible for presenting the different data abstractions (blobs, tables, and queues), organizing those objects into a namespace so they can be located by their keys, and replicating objects to remote datacenters for disaster recovery. Below the partition layer, the stream layer organizes data updates into blocks to be written to underlying storage, and ensures that these blocks are replicated to different devices across different fault domains. In each fault domain, data is is written to stream nodes which are independently responsible for storing their own data. I will now describe the mechanisms by which each layer performs these functions.

At the bottom of Figure 2.1 I have formed a valid URL to write a blob of data to Azure [2]. This request is sent over the internet to the load balancing layer, which is made simple to avoid acting as a bottleneck. Based on physical location, a load balancer routes the request to the partition layer below. Conceptually, the payload for this new blob of data will be written to

[2]Setting aside the authentication and authorization requirements.

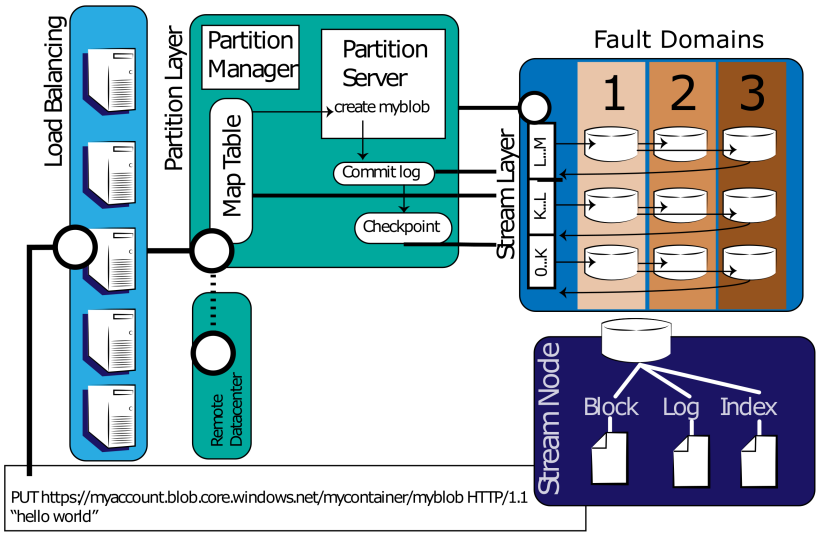

Figure 2.1: Microsoft Azure Storage: A simplified example

disk and a record of the object will be recorded in a lookup table. However, because a lookup table for trillions of objects would be impractical, it must be split across the nodes in the system. To identify the location in the partition table that will be reserved for this object, an index in the *map table* is created. That table directs the request to the appropriate partition server, which writes the payload, and logs an entry of the blob's creation in a *commit log*. Periodically, the commit log is collapsed into a *checkpoint* to save space, and any stale entries are collected. Each of these four objects: The map table, commit log, checkpoint, and the blob payload itself are written to the next layer in the stack – the stream layer. In addition, the partition layer also replicates the request to another datacenter for the purpose of

recovery in the event of a datacenter-wide failure, thus at a global scale, every operation in this example is repeated twice.

At the stream layer, each write to each of these objects is split into a number of updates, and those updates are organized into blocks for storage to disk. Small writes may fit in a single update, while large writes may be split across many. Blocks are replicated across three fault domains, which are logical organizations of storage devices that partition the cluster into regions within which there are no correlated sources of failure. Among other restrictions, this means that each replica must be written to a different node in the system.

These stream nodes each maintain their own blocks, and block locations are written to an index so they can be found later. In addition, a copy of the block is written to a log as an optimization. Usually this log write is faster than writing the index and the block, which allows the stream node to acknowledge the write as complete once is is logged. The log is not indexed, but it can be replayed to rebuild the index and original block in the event of failure. These three objects at the stream node (the block, index, and log) are written to files within the the Windows NT file system (NTFS). NTFS is itself a complex storage artifact, which is to say that these writes may require journaling and metadata and inode updates, in addition to writing the actual data. The tracing and analysis in my thesis is all based on NTFS, which I discuss in more depth in Chapter 3.

In sum, the single original write request, which may have been issued to Azure's URL from a user's phone or browser, and likely without their explicit knowledge, results is a tremendous cascade of subsequent requests.

It touches potentially four objects at the partition layer. Each of those may be further subdivided at the stream layer, which then replicates all of the resulting requests 3 ways. Each of those replicas can yield three more writes at the stream nodes, which in turn can create subsequently more requests within NTFS. Furthermore, each point of the system's design represents a decision made by a file system developer based on workload assumptions, if not direct measurement. For example, consider the checkpoint rate of the commit log at the partition layer. An overly aggressive checkpoint rate yields write amplification through the system. On the other hand, infrequent checkpointing lengthens the log, which can hamper recovery.

This complexity in Microsoft Azure Storage is largely unavoidable because it comes as a direct response to the burdens of its workloads. In 2014, it serviced 3 million requests per second, across 30 trillion user objects [Gur14]. However, it is not unique in this regard. Many other cloud storage systems are similarly complex, yet are completely different in their implementation. Facebook's Haystack is designed for servicing petabytes of immutable photos [BKL+10]. GoogleFS is a general purpose scalable cluster file system used internally for many of Google's services [GGtL03]. Besides these and many other cloud storage examples, modern enterprise storage arrays are also generally implemented as a cluster of servers which coordinate to spread load and distribute data, which means they share most of these same drivers in system complexity. Increasingly, these systems are also servicing millions of requests per second [CWM+14], just like Microsoft Azure Storage.

By comparison, consider the canonical Berkeley Fast File System (FFS),

which was originally evaluated on a single system with 920 Megabytes of formatted space and a 983 kilobyte maximum throughput [MJLF84]. This is no straw man: FFS remains quite influential, so much so that many of the insights drawn from that work continue to be used today. The 4 kilobyte block size employed by most file systems is a direct result from FFS, as is the practice of allocating inodes statically in each cylinder group, which makes inode count a function of disk size. These insights were not obvious at the time; they were drawn from measurement and analysis of a then complex system. However, the parameters in FFS and their expected impacts were determined based on the measurement of a live user system with characteristics that bear little resemblance to storage systems today.

Microsoft Azure Storage is radically more complex than FFS. Even when one understands the structure and design, the overall behaviour under a load of millions of requests per second spread over exabytes of data in trillions of objects is impossible to intuit accurately. However, as was true for FFS, measurement is required to make good decisions in all storage systems, across all storage domains. The complexity of modern storage systems not only calls the validity of legacy file system choices into question, it also challenges our ability to even consider their impacts. This is because performing analysis on large-scale high-throughput systems with complex request inter-dependency and large features sets is a difficult undertaking. In the following chapter, I provide background on tracing and analysis. I describe how the tracing in this thesis was performed, how it has been done in the past, and how that relates to file system design and architecture.

Chapter 3

Background

There is a long history of tracing and analysis of storage systems, both as a research endeavour to provide general insights into storage system behaviour [OCH+85, BHK+91, Vog99, WDQ+12, DB99, ABDL07] and as a practice to evaluate a specific idea or argument [AADAD09, AAADAD11, TMB+12, BGU+09]. In addition to the specific findings I will detail in the following chapters, this thesis builds upon that analytic tradition, while simultaneously demonstrating new techniques to facilitate more frequent, larger scale, and more detailed tracing and analysis practices. This chapter serves to contextualize my contribution and provide the background to understand the tools that I have developed and how they have been applied to working systems.

I have argued that storage systems are complex, and that useful information about their operation is often obscured. To understand why this occurs, how tracing and analysis can help, and the limits of this approach, one must understand how storage systems process requests. In Section 3.1 I begin by describing, in the general sense, problems that tracing tools face as they relate to the storage stack. I consider where tracing might and should occur, how traces are gathered, and potential pitfalls. In Section 3.2 I simi-

larly describe the choices one is presented with when analyzing the on-disk data that results from the use of the storage stack over time. In Section 3.3 I describe the organization and architecture of the Microsoft Windows storage stack. I also highlight where Windows employs designs that differ from Linux and other operating systems. Finally, in Section 3.4 I describe the tracing and analysis I have done as part of the case studies in this thesis and compare my work to related research.

3.1 Storage Tracing

Storage traces, such as "Measurement and analysis of large-scale network file system workloads" from 2008 [LPGM08], are built from recording the activity that passes through some point in the storage stack. Since traces focus on the active use of the system, they include details about data that is accessed over the duration of the trace, but typically do not include information about files that are not accessed, and typically do not include the contents of files or of file system metadata. Typically tracing involves loading a software module or otherwise modifying the storage stack in order to passively monitor the requests. However there are several decisions that must be made when performing such a trace. In this section I describe what can be learned from tracing storage systems and detail some of the concerns common to storage system traces.

3.1.1 Applications of Workload Traces

Traces can be applied to improve storage systems in a number of ways. For the comparative study of systems, they can be replayed either by informing the creation of a synthetic workload generator, or by direct replay. For example: Tarasov et al. used block-level traces of build servers and online transaction processing servers to generate replayable workload generators in 2012 [TKM+12]. Traces can also be studied to produce statistics and characteristics of the workload in order to guide system design. For example: A trace gathered by Narayanan et al. in 2008 was used to detect hot-spots in file accesses within a storage cluster in order to divert workload to less loaded machines [NDT+08]. When applied to existing systems, traces can often identify performance bottlenecks, misconfigurations, or flaws by highlighting areas of poor performance or wasted effort [HBD+14]. As storage systems grow more complex and workloads become more intensive, there are more opportunities for such inefficiencies to go unnoticed.

Each of these applications of tracing is featured in the rightmost column of Table 3.1, which lists the data associated with individual entries in most storage traces. The leftmost column lists the corresponding characteristic of the workload that is typically accessible in a trace of the system. Of course, these results can be made more valuable by aggregating them. For example, one can determine and compare I/O operations per second of different systems, or read/write percentages of common workloads. The middle column describes the approximate layer of the storage stack (block or file) where that characteristic is typically available. Some characteristics are available

Characteristic	Layer	Example Application
disk offset	block	Characterizes linearity of access to disk.
request size	both	Measures workload throughput for evaluation.
issue time	both	Ensure correct timing in a replayable workload.
req. latency	both	Measures performance of underlying storage system.
read/write	both	Sets optimization priorities based on request.
request type	file	Open-to-close duration provides file lifetimes.
target file	file	Access distribution is used in workload modelling.
file offset	file	Linearity of access informs prefetching systems.
source app.	file	Attributing problematic workloads to applications.
access flags	file	Flags (sync, temp) give hints for workload prediction.
cache status	file	Analyzes effectiveness of page cache.

Table 3.1: Block vs. file characteristics. The different characteristics typically measurable at block and file-layers and how they might be used.

at both file- and block-levels, though their interpretation would be different depending on the level in question. Further, tracing typically only provides insight into one point of the stack, so not all information will be available in every trace. I return to this table in the next section, when discussing the relative merits of tracing at different points in the stack.

3.1.2 Trace Altitude

The modularity of the storage stacks in most file systems afford multiple locations where one can trace, and each yields different results. The higher levels of the file system are useful because they provide access to the relatively rich file system API. At this layer, for example, distinctions between file data and file metadata access are apparent, and one can generate a trace suitable for replay against a file system. As shown in Table 3.1, there is clearly richer semantic information at the file-layer; however, the information at the block-layer is generally no less useful. The logical disk

offsets where data actually lies are critical to understanding the linearity of workloads, which can have a huge impact on disk performance, but is typically unavailable above the block-layer. Furthermore, because requests are frequently added and removed throughout the storage stack, the sum of requests that are ultimately issued to hardware is only visible at the lowest layers of the stack.

Another place to trace is above the file system entirely – at the application-level. Application-level tracing may provide some increased ability to understand semantic patterns in application workloads [YBG+10], but also removes any insight into file system or disk operation. Most commonly, such traces are used to build synthetic workload generators or replayable trace applications. Furthermore application traces are usually employed when there is one critical application running on a system, such as an NFS server, that will be generating most of the storage requests.

3.1.3 Cache Considerations

The other primary concern with respect to tracing is the trace point in relation to memory caches. For example, when building a replayable trace, it is often helpful to both trace and replay below the cache so as to normalize the effect of caching between the original host and the system subject to replay. In contrast, some traces such as "File System Usage in Windows 4.0" from 1999 [Vog99], have included trace information above the cache in order to study the behaviour of the cache management sub-system itself.

Block device drivers generally operate below the cache, and although the unified cache in Linux does include a buffer cache that is capable of

block-level caching, in practice most population of that cache is done at the file-level. With the Windows Cache Manager, I/O is visible to all filters above the Virtual File System (VFS)-layer whether the request is cached or not, and so tracing at this layer is generally parallel to the cache – neither above nor below. I will discuss this arragement in more depth in Section 3.3.

3.2 Storage Analysis

I now turn to discuss storage system analysis, which differs considerably from active tracing. Whereas traces passively observe live activity from a running system, a storage metadata or content study generally queries the storage system in order to discern the state that prior workloads have driven it into. There are a wide range of characteristics of storage systems that can be studied from this perspective, but most attributes of on-disk data can can broken into four categories: First, information about file system metadata, which is the data that a file system stores about each of its files and the file system generally. Second, the organization of files within the file system, such how they are placed within directories. Third, where data ultimately is placed to disk. And finally, analysis of the data within the files themselves. This section describes these characteristics in more depth and explores how they can best be analyzed and what can be learned from them.

3.2.1 The Value of Storage Analysis

The structure and content of a file system provides many opportunities for performing interesting analysis. Table 3.2 provides a small sample of

Example Attribute	Attribute Type
File size	File Attribute
File flags	File Attribute
File type	File Attribute
Time stamps	File Attribute
Space Utilization	FS Attribute
File name	Namespace Organization
Path	Namespace Organization
Hard links	Namespace Organization
Data Fragments	On-Disk Location
Compressability	Data Attribute
Deduplicability	Data Attribute

Table 3.2: Metadata examples. Examples of metadata that can be included in a storage system analysis.

attributes that are typically available, and a categorization of the type of attributes according to the 4 categories above.

In the past, analyses that have drawn on findings like those in Table 3.2 have leveraged better understanding of real world namespace organization to build synthetic file system generators [AADAD09], which mimic the layout of real file systems but can be parameterized and used to test new applications or storage systems. Similar results in the file attributes domain can also be used to inform the structure of file systems, for example, guiding designers to correctly optimize for the ratio of small to large files. The distribution of file sizes is a useful statistic when designing a file system because it informs, for example, the portion of files can be stored directly with the file metadata (in Windows, such a file can exist entirely in the Master File Table) and the portion of files that will need an indirect reference to a block of on-disk data. Similarly, knowing which file types tend to be large is useful when allocating contiguous space on the disk. Better characterizations of on-

disk data placement influences research in data layout optimization [HSY05, BGU+09]. Understanding the structure of data itself has many applications as well, including the inspiration and evaluation of new data management techniques such as compression and deduplication [SPGR08, CGC11].

As file systems grow larger in capacity and the feature sets they expose grow more complex, understanding these attributes becomes more important. For example, very large systems are often driven to combine storage resources through virtualization, and this creates opportunities for shared caching or redundancy eliminating storage, however the effectiveness of such techniques depends on the similarity of data across different machines. Understanding the real world applicability of such a mechanism prior to investing in its construction is only possible with tracing and analysis of existing systems.

3.2.2 Storage Analysis Considerations

Unlike storage stack tracing, on-disk data analysis is almost always performed at the file-level, where the semantic information associated with files can provide structure to the data. Presuming one finds a suitable collection of file systems to measure, storage system analysis typically struggles with two mechanical issues: scanning the file system is *time consuming*, and the resulting datasets can be *very large.*

The severity of both issues depend on what is actually read. If only file metadata is considered the data can typically be gathered in minutes and stored on the order of megabytes per file system investigated. This can be done in Windows by reading the Master File Table or the inode tables in

Linux's Ext2/3 file systems. If the study requires processing file data, as the analysis in this thesis does, the process takes considerably longer. In either case, the process takes long enough that the file system is likely to be changing while the data is gathered. For many types of analysis, this is undesirable, because it means the file system is not in a crash consistent state as it is read, and thus may include inconsistencies. If file system snapshots are available, as they increasingly are in modern file systems [RBM13, Cor10a], they may be used to create a crash consistent file system image. As I show in Section 4.1.2, where I describe my own file system analysis framework, Windows can create a fully consistent view of the file system for many applications. If snapshots are used in this manner, care must be taken that they themselves are not treated as subjects of the data analysis.

A second concern with a file system study is where the data generated in the study is stored. Storing the data directly to the file system itself must change the file system. Alternatively, storing the data on a remote network storage target, particularly if the dataset is large, risks data loss due to unavailability of the network store. One solution is to eliminate entries associated with storing of data from the trace in post processing, which removes much of the effects of tracing, though does still potentially change the physical layout of the data on disk somewhat.

3.3 The Windows Storage Stack

In this section I describe the Windows storage stack because it is the subject of the tracing and analysis in this thesis. Tracing in Windows is challenging

because of the breadth and complexity of its APIs. However, it is also rewarding because Windows represents a large installed base of computer systems.

Like most storage systems, The Windows storage stack is structured as a collection of layered software modules that each transform storage requests from the high-level interface enjoyed by applications progressively down to the raw access provided by the disk drive itself. This layering serves two purposes. First, it simplifies the storage system. At each layer a developer can restrict their concerns primarily to the operations and semantics required by requests at that layer. Second, it provides extensibility, in that most operating systems provide some mechanism for adding multiple file systems and other custom software modules to the storage stack. Both Linux and Microsoft Windows provide this support by allowing administrators to load modular kernel-mode drivers that manipulate storage requests.

An illustration of the structure of the Windows storage stack is shown in Figure 3.1. For the sake of simplicity I have limited the discussion to components relevant to this thesis. Through the remainder of this section I will describe the Windows storage stack from the bottom to the top. A more comprehensive discussion can be found in [RSI12].

3.3.1 Device Management and the Block-Layer

At the bottom of the storage stack in Figure 3.1 is the block-layer, which operates very much like a simplified interface to an underlying disk device. Each request is either a read or a write to a logical block address on the device and is aligned to a 512KB boundary. However, requests are almost

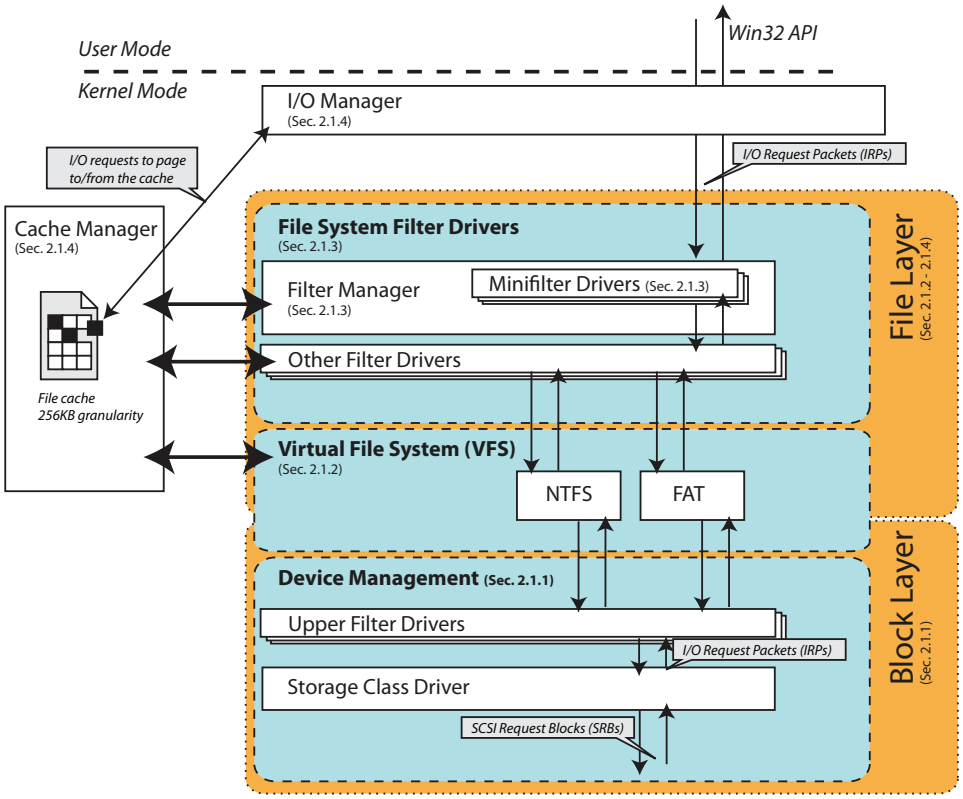

Figure 3.1: The Microsoft Windows storage stack.

always aligned to a 4KB boundary and a multiple of 4KB in size. There are no files or directories at this layer. The stack is also bidirectional, in that each driver can register to see requests issued (down the stack), and separately to see them completed (up the stack). Though the programming interface is quite different, this layer is analogous in its position and power to the Device Mapper in Linux. Every software module at this layer serves to prepare requests for the underlying hardware, and are ordered in the stack in the layer that they are loaded into the operating system.

The lowest layer of the stack I consider in depth is where *upper filter*

drivers are installed, but as the name suggests there are layers lower. Upper filter drivers operate on a data structure called an I/O Request Packet (IRP) which encapsulates an I/O request through most of the storage stack. Below this point, a storage class driver, such as the generic iSCSI driver, transforms the IRPs into other structures. Less conveniently for the sake of tracing, the storage class driver may also split and merge requests. This means that while upper filter drivers see patterns of requests that closely match those of the underlying device, they do not have visibility into some operations performed at lowest layers of the stack.

3.3.2 The Virtual File System Layer

As shown in Figure 3.1, above the block-layer Windows supports a virtual file system interface in which multiple file systems can be installed, as does Linux. However, most Windows workstations use NTFS as their primary file system, [3] and as NTFS is a commercial off-the-shelf closed-source product, insight into the specific transformations at this layer is challenging. Generally, like UNIX file systems, NTFS transforms requests upon files, directories, and metadata into the relatively simple block-level requests of the layer below.

3.3.3 Filter Manager

Every layer above the VFS operates on the file API. Like the layers below, the IRP structure is used here, but this interface is considerably larger and

[3] Unlike Linux, where there are several commonly used file systems, there is a single file system used exclusively on almost every non-removable drive in every Windows workstation - NTFS.

richer than the block-layers. It includes read, write, file open and close, and create and delete. Among other filter drivers that may be installed on a Windows workstation, Figure 3.1 highlights the filter manager, which exposes a framework for writing *minifilter drivers* (minifilters). Like filter drivers generally, minifilters register routines that process requests made by Windows programs and users. Unlike filter drivers, minifilters feature a simplified interface, load in a deterministic order, and can be dynamically unloaded. Each routine may forward requests it receives down the stack, or complete them without passing them on. They may also issue new requests, which I will discuss further in the next section.

Each minifilter driver is assigned a unique altitude, which is a number that determines when it is loaded and where it resides in the storage stack. For the purposes of tracing, one would conventionally use an altitude in the *FSFilter Activity Monitor* group, which includes altitude values 360000 to 389999 for "drivers that observe and report on file I/O" [Cor14b]. There are 22 other altitude groups available, but the Activity Monitor group is quite high in the order. This relatively high altitude means that traces obtained at this level closely match the original stream of requests made by applications, because they have not yet been transformed by drivers of lower altitude.

3.3.4 I/O and Cache Managers

In Figure 3.1, two more components lie above and beside the filter driver layer, respectively. The filter drivers receive requests from the *I/O manager*, which is the arbiter of all requests onto the storage stack. The I/O manager acts as a central point of dispatch to deliver IRPs to the correct layer of

the stack in the correct order. This is true even if the requests originate from drivers themselves, in which case the I/O is performed recursively. Recursive I/O passes back through the same driver again on its way from the I/O manager down the stack. Like non-recursive I/O, it also triggers completion events on the way back up the stack. This design has many implications, among them that file requests issued by the file system itself will ultimately be seen above the file system by the routines of file system filters.

Another important consequence of this design is that, unlike a canonical storage stack which has a singular layer that services cached requests, the cache manager in the Windows storage stack sits aside the storage stack. Requests that are serviced by the cache manager still traverse the entire stack through to the VFS-layer, but do so with a flag denoting the operation as one that operates on the cache. Similarly, paging operations that move data out of the cache traverse the stack the same way (through the file-layer), but with different flags. As a result file system filter drivers can all see the paging activity caused by the cache manager.

This organization also means that any I/O created by a tracing process must be reissued through the I/O manager and thus will be visible to the layer that issued it. This can be an issue if a tracing system must issue I/O requests to write the tracing logs to disk, or if the the tracing driver must issue new requests to, for example, determine metadata such as a file name. Generally such artifacts of tracing are best removed from a trace.

3.3.5 Discussion

Storage stacks generally, and the storage stack used in Windows workstations specifically, are complex and extensible. Each layer of the stack obscures the context of operations that pass both above and below it, which makes understanding the I/O requests that flow through the entirety of a storage system difficult to characterize. Furthermore, requests may be batched, delayed, merged, and/or split by the filter drivers, the file system, the block layer, or any other layer of the stack. Furthermore, this manipulation of requests often leads to read- and write-ordering dependencies between requests that the software must manage. As a result, the behaviour of a storage system under a real workload is complex and difficult to understand. Further, for any given workload, the behaviour at different layers of the stack may be quite different. This means that intuiting storage system behaviour without a trace is extremely challenging, and yet tracing is used in a vast minority of research storage system evaluations [TBZS11]. The problem is amplified with heavier load and with the size of the storage stack API. This also means that no small number of traces can characterize a broad pool of workloads at multiple layers of the stack. Part of my thesis, as described in Chapter 1, is that the benefits of understanding storage system response to workloads are significant, and that more tracing in more depth is required to facilitate that understanding.

In the following two sections, I discuss how to approach instrumentation of the storage stack and how to analyze the effects of requests on a file system. I then discuss existing research as it relates to the tracing and

analysis I will present in this thesis. In Chapter 4, which follows, I describe the tools I have written to collect data and how those tools interact with the storage stack.

3.4 The Traces and Analysis in This Thesis

Next, I will describe the traces and analysis conducted to support this thesis; I present two significant case studies. The first is an on-disk data study of file system content gathered in 2009 at Microsoft Corporation. The second is a trace of storage system activity at the University of British Columbia in 2010. Collectively, these studies characterize behaviour at both file- and block-layers of the stack, include most file and file system metadata, the content of files themselves, and describe detailed workloads at scales up to one week, as well as the changes in on-disk content over the course of a month. In this section I describe these traces as they relate to other research.

3.4.1 File System Content Study

My study of file system contents and metadata includes nearly 1000 machines which were scanned at Microsoft Corporation in Redmond, Washington over 6 weeks. The bulk of my analysis in this thesis focuses on 857 file systems spanning 162 terabytes of disk over 4 weeks, selected because I have reliable data from each of these systems for every week in that period. The results includes data and metadata from a broad group of employees, including software developers, management, sales & marketing, support, writers and legal staff.

These results are notable in three ways. First, as I show in Table 3.3 below, analysis of file system metadata is rare. Mine is the only published study in the past decade to provide a characterization of deployed workstation file system metadata. Second, my study considers snapshots of file system content on a weekly basis for four weeks. This means that in addition to characterizing large scale trends in file system evolution by comparing my results to those of similar studies, my dataset can identify shorter term trends such as the portion of files that are unmodified week-to-week. Finally, this study is the first to measure and record the content of the files within a large set of general purpose machines. This allowed me to draw conclusions about the rate of file modification as well as to compare several approaches to a storage feature called deduplication. Deduplication is a process by which two or more large regions of identical data can be stored once, in order to reduce data sizes. I discuss deduplication in more depth in Chapter 5.

The impact of working at this scale was significant, as the highly compressed logs gathered in this study comprised more than 4 TB of data. The analysis of this data, particularly that which is described in Chapter 5, required identifying and counting small common sub-strings in that corpus, which is a time consuming and disk and memory intensive workload. Microsoft Corporation has released an anonymized version of this data to be housed at the SNIA IOTTA storage trace repository for public use [Ass13]. I consider the results of this study throughout this thesis, in Chapters 5, 6, and 7.

Study Source	Year of Study	Metadata or Data	Count: FS = File system DC = Data center Dataset = A collection of subdirectories	Data Content Analyzed	Total Compressed Study Size	Duration
Douglis [WDQ+12]	2012	both[2]	8 FSes[3]	682 TB	Unknown	1 week
Lu [LCGC12]	2012	data	4 Datasets	10 TB	Unknown	Point-in-time
Park [PL10]	2010	data	6 Datasets	150 GB	Unknown	Point-in-time
Meyer [MB11]	**2009**	**both**	**3,500 FSes**	**162 TB**	**4 TB+**	**Weekly, 4 weeks**
Zao [ZLP08]	2008	data	2 DCs	61 TB	Unknown	Daily, 31 days
Policroniades [PP04]	2004	data	4 Datasets	200 GB	Unknown	Point-in-time
Kulkarni [KDLT04]	2004	data	5 Datasets	8 GB	Unknown	Point-in-time
Aggrawal [ABDL07]	2000-04	metadata	70,000 FSes	N/A	91.1 GB	Yearly, 5 years
Douceur [DB99]	1998	metdata	10,000 FSes	N/A	3.7 GB	Point-in-time

Table 3.3: File system studies. Significant published studies of file-system data and metadata published since 1999.

There are relatively few large scale studies of on-disk metadata or data in the published literature. In Table 3.3 I list the significant file system metadata and deduplication content studies published since 1999.

In 1999 [DB99] and then in subsequent years until 2004 [ABDL07] very large pools of workstations were analyzed for file system metadata. Relative to the work in this thesis, those studies include a larger number of workstations because they did not have the burden of analyzing on-disk content, only metadata. These studies are extremely valuable in the context of the metadata results in this thesis, because they provide an older snapshot of file system usage against which to compare.

There are also relatively few analyses of deduplication ratios, and the ones that exist suffer from several common limitations. Policroniades [PP04] studied a small corpus of data from a variety of sources in 2004 (200GB, about 0.1% the size included in this thesis) which, contrary to my results, found little whole file deduplication, and a modest difference between fixed block and content-based chunking. Kulkarni [KDLT04] used an even smaller corpus in 2004 (8GB) to analyze fixed block deduplication, compression, and delta-encoding. A similar study was performed by Park [PL10] on a dataset of 25GB in 2010. Others have attempted to measure deduplication rate against small corpuses of synthetically generated data [JM09], which may be more easily scaled to large sizes, but is of questionable accuracy.

[2]Study included data pertaining to file lifetimes within the deduplication system itself, not the user metadata subject to backup.

[3]Over 10,000 machines were used for metadata analysis based on autosupport logs, whereas 8 datasets were used for deduplication analysis.

The largest studies of real-world deduplication rates are performed by deduplication vendors themselves [ZLP08, WDQ+12]. These rates provide useful data points, but also reflect self-selection in that they measure a corpus of users who have decided that they would benefit from deduplication. In addition, it is difficult to distinguish the innate deduplicability of the data from the deduplicability of the workload. For an in-line stream-based deduplication system there is little distinction between the two. If a user has a single file system that at any point in time has just 1% data duplicatation, but one then creates a full backup of that system 100 times, the deduplication rate would be reported by commercial vendors as approximately 100x. Results in this manner present a useful insight into the workloads applied to deduplication systems (how many times files are replicated for the purposes of backup), but do not necessarily reflect the potential for reducing the overall work that a deduplication system must do, for example, by employing alternate approaches to deduplication which I will discuss in Chapter 5. Relative to the results in this thesis, Wallace, Douglis, et al. [WDQ+12] published after (and in some sense perhaps, in response to) the work I describe here. They found much higher deduplication rates across a larger corpus of data, but in many cases this is a question of parameterization rather than data content – the data was simply backed up more times in practice than the 4x assumed in my study. Regardless, their insights into the workloads that drive stream-based deduplication systems are unique and extremely valuable.

With respect to Table 3.3, compressed study sizes are not published for many of these datasets, and the datasets themselves are not publicly

available, so their study size, shown in the second column from the right is unknown. In each such case they are likely to be much smaller in terms of the size of the data analyzed than this thesis. Even in the case of the Douglis [WDQ+12], where 10,000 disk images were considered, only file metadata was collected over that set, and from the 8 very large file systems that made up the 682 TB of data content studies, only the aggregate results of deduplication were retrieved. In some ways this is a good thing, as smaller study results are easier to manipulate. However, it also precludes any further in-depth analysis if the data from such a study was to be made public, and means that those efforts did not face the same challenges associated with processing large datasets.

A few systems have approached the problem of deduplication in primary storage [DGH+09, UAA+10], and have used largely synthetic workloads in their evaluation. While the compressability results appear roughly in line with my own, it has yet to be seen if aggressive deduplication in the data path is practical from a performance perspective. In "Insights for Data Reduction in Primary Storage: A Practical Analysis" [LCGC12], published after the work in this thesis, researchers roughly confirmed many of my findings with respect to deduplication rate.

Both commercial and academic deduplication systems should benefit from the results of my file system study [LEB13, KR10, CAVL09, BCGD00, LEB+09, KU10] in being able to make more informed choices about the trade offs associated with deduplication. Similarly, these efforts may spur continued research into developing accurate models and benchmarks for deduplication analysis, which has only just started [TMB+12].

3.4.2 Storage Stack Tracing

There are more publicly available traces of real Linux or Windows system workloads than there are disk-content studies. However, there are still surprisingly few, and little about tracing methodology has changed since Vogel's analysis of Windows behaviour in 1999 [Vog99]. I have attempted to improve the state of knowledge and practice of file system workloads by publishing the traces in this thesis, and extending live workload collection practices by including richer file system and block-level metadata. I have also proposed simplifying the practice of creating and collecting these traces by leveraging increasingly common virtualized environments, which was cited in 1999 as one of the major challenges in performing this type of research [Vog99].

My workload study consists of a week-long trace of all storage traffic from a production deployment of 55 Windows Vista desktops in an executive and administrative office of a large public organization. The traces were gathered by University of British Columbia's IT department, UBC IT, under my supervision and using tools I developed. My trace is shown in bold in Table 3.4 along with all relevant traces of storage system activity published since Vogel's 1998 trace of Windows workstations published in 1999 [Vog99]. Traces are sorted by year they were conducted.

My study is notable in that it includes simultaneous tracing of the file- and block-levels of the operating system. [4] This has allowed me for the

[4]Here I am tracing a virtualized environment, and have chosen to trace the guest operating system, as opposed to the host. This means I am tracing the logical block-level as seen by the guest, which will be quite different from the physical block-layer in the host operating system. In the context of this trace I use the term "disk" throughout this thesis

first time to relate disk access to the region of the file system for which the access is directed. I have also studied NTFS file systems in a relatively unexplored context - that of a virtual desktop deployment. Thus, the block- and file-layer shown in Table 3.4 refers to the virtualized guest as opposed to the virtualized host. Finally, the tracing in this study is facilitated by the virtualized environment, wherein a tracing driver could be installed and tested once, then deployed to all machines in the cluster instantly. I consider this study throughout this thesis, with the bulk of workload analysis in Chapters 6 and 7.

to refer to the virtual disk as seen by the guest operating system.

Trace Source	Year of Trace	Layer	Request Count	Trace Size [5]	Machine Count	Duration
Facebook [HBD+14]	2013	Application (HBase)	5.2B	116 GB	Unknown	8 days
Yahoo [ALR+12]	2012	FS Metadata (Hdfs)	Unknown	Unknown	4000+	1 month
M45, CMU [RKBH13]	2010-12	Application (Hadoop)	100K	Unknown	473	20 months
UBC [SMW+10]	**2010**	**Block+FS**	**300M+**	**75GB**	**55**	**8 days**
FIU srcmap [VKUR10]	2008-09	Block	26.7M	14.6 GB	1	1 month
MS Prod Other [Ass14a]	2008	Block	163M	2.6 GB	6	1 day
FIU home [KR10]	2008	Block	18M	2.1 GB	1	20 days
FIU mail [KR10]	2008	Block	443M	11.6 GB	1	29 days
FIU web [KR10]	2008	Block	14M	351 MB	1	29 days
MS Prod. Build [Ass14a]	2007	Block	1.2B	13.0 GB	1	18 hours
MS Exchange [Ass14a]	2007	Block	64M	1.4 GB	1	1 day
Animation07 [And09]	2007	Application (NFS)	144B	2 TB	1-50 [6]	9 months [7]
NetApp2 [LPGM08]	2007	Application (CIFS)	352M	1.5 TB [8]	1	97 days
NetApp1 [LPGM08]	2007	Application (CIFS)	228M	750 GB [8]	1	65 days
MSR [NDT+08]	2007	Block	434M	29.0 GB	13	7 days
Animation03 [And09]	2003	Application (NFS)	19B	798 GB	1-50 [6]	6 months [7]
Harvard email2 [Ass14a]	2003	Application (NFS)	2.1B	32.5 GB	3	7 weeks
Harvard email [Ass14a]	2002	Application (NFS)	1.7B	24.7 GB	3	6 weeks
Harvard home04 [Ass14a]	2001	Application (NFS)	295M	2.5 GB	1	4 weeks
Harvard home03 [Ass14a]	2001	Application (NFS)	2.1B	30.8 GB	1	14 weeks
Harvard home02 [ELMS03]	2001	Application (NFS)	3.3B	49.0 GB	1	16 weeks
HP instructional [RLA00]	2000	FS	318M	68.9 GB [9]	19	31 days
HP research [RLA00]	2000	FS	112M	24.3GB [9]	13	31 days
HP desktops [RLA00]	2000	FS	145M	31.4GB [9]	8	31 days
Berkeley PCs [HS03, ZS99]	1992-00	Block	13M	5.8 GB	18	45 days [7]
Cornell PCs [Vog99]	1998	FS	190M+	19 GB	45	4 weeks

Table 3.4: Storage traces. Significant published traces of live storage system activity published since 1999.

In comparing my trace to other work, one of the highest quality enterprise workload traces is from the evaluation of write off-loading [NDT+08, NDR08] which used data gathered in 2007. This trace contains a collection of several isolated independent servers, each traced over a relatively short period. My overall workload is similar in size and duration, but focuses on a set of workstations operating concurrently. Still, much of the basic characterization of requests is similar, for example both show an approximately 2:1 write to read ratio and bursty access. In both cases, the data was used to motivate systems that attempted to address peaks in workload, albeit through different mechanisms, and both traces gather information at the block-layer, though my trace simultaneously captures file-level information. There have been few published traces capturing file-level information, and the other block-level traces have been categorically smaller, with the exception of the trace of the mail server at Florida International University (FIU), which is a significant, but very homogeneous and specialized workload from a single server. Like my trace, the FIU traces have at times traced virtual machines, capturing file- or block-level requests as they appear to the OS, but not the underlying virtual host system.

Since 2001 there have been a number of traces of the NFS and CIFS protocols that can be largely clustered into three bodies of work: The Har-

[5] Compressed, unless otherwise noted.

[6] In some time periods, 50 caches of the central NFS server may be seen as additional servers, in other periods only a single server downstream of the cache is seen.

[7] Tracing was enabled and disabled periodically over this time.

[8] Uncompressed TCP dumps.

[9] Scaled as percentage of requests in 150GB aggregate workload.

vard traces from 2001 [ELMS03], the NetApp traces from 2007 [LPGM08], and the traces of an animation cluster in 2003 and 2007 [And09]. Of these, the animation traces stand out for their size. However, due to the size, the results only cover periods of times for which the trace recording server was not full, which was a minority of the time. The result is a non-contiguous collection of large and intense workload traces, starting and stopping at seemingly random points with different collections of traces using different methodologies and trace settings. Still this is a useful trace for many purposes, and is one of the very few public traces that captures an intense workload. Naturally, these protocol-level traces capture the workload as it appears over the network, but not at the underlying device or file system.

Many traces are now more than a decade old and are of questionable relevance today [ZS99, RLA00, HS03, ELMS03]. Still others have made interesting analytic contributions by tracing synthetic workloads [HDV+12, GKA09, WXH+04, THKZ13]. Since publishing the results in this thesis, some have called for increased attention to system workloads in a variety of environments from data centers [RKBH13] to desktop computers [HDV+12], and each has contributed new results. However, the storage landscape continues to expand quickly. As has been pointed out by Tarasov recently in the context of virtualized NAS storage [THKZ13], our production environments have outpaced many older workload models. One potential solution, the one I call for in this thesis, is to perform workload traces frequently, in a wider variety of environments, over larger sets of data, and digging deeper into richer information gathering. My tracing dataset makes contributions in these areas, but suffers primarily from being small, even as it is

relatively similar in size to most comparable general purpose workstation traces [Vog99, NDT+08, LPGM08]. In the future, the scalable methods of deployment and analysis I describe in this thesis may lead to still larger studies.

3.4.3 Improving the State of Storage Analysis

One application of system traces and file system models that I have discussed is applying them toward improving storage system evaluation. Researchers have noted many long-standing challenges associated with system evaluation [TJWZ]. Recently Tarrisov et al. warned that without changing our evaluative criteria, "our papers are destined to provide incomparable point answers to subtle and complex questions." [TBZS11] To address these concerns, systems have been developed to facilitate analysis in the past [WJK+05, AWZ04], or that suggest modifications to current practices [SS97]. My approach is complementary to these efforts in that I have significantly expanded the body of findings, and raw data, available to researchers while offering several new methods for collecting and processing future studies of this sort.

By adding to the published body of knowledge about use and behaviour of live systems under real workloads, my dataset helps facilitate better system modelling [AADAD09, AAADAD11], replay [TKM+12], and understanding of specific features such as deduplication [TMB+12, HMN+12]. Researchers have continued to make these methods more accurate and deployable, for example, by extending tracing to include causal links between requests both in the storage stack [JWZ05, WHADAD13, SAGL06] and

elsewhere [BCG+07]. These approaches, to the extent possible, are good candidates for extending my work and expanding future studies. Other examples include efforts to diagnose problems based on performance anomalies in a distributed system [KTGN10, SZDR+11]. Better and more comprehensive tracing can only improve these efforts by providing more opportunities to refine detection heuristics and richer information to consider at each trace point.

As I will discuss in the following chapter (Chapter 4), one way in which my workload traces differ from most is that the trace of UBC workload includes information typically associated with higher-level processes (e.g., originating file name), but collected below the page cache where the underlying storage system extends most of its effort. This approach mirrors efforts in the distributed systems community to understand request flows through large complex systems [BDIM04, LGW+08, AMW+03, CAK+04, GAM+07, CCZ07, KJ08] by tracking requests. Similar efforts have been made in the storage community [BBU+09], but not within isolated storage stacks, and not for the purposes of detailed trace analysis.

A different approach to understanding system behaviour is analyzing request logs, where researchers have made efforts to draw deeper knowledge from logs and autonomously expand their content [YZP+11, YMX+10, JHP+09]. This effort can be seen as a parallel approach, wherein the entire space of the executing service can be mined for more information, but one that usually focuses on anomalous behaviour. My approach has been specific to storage concerns, where my system tracing work has focused specifically on the most common request flows. Further, my research on on-disk struc-

tures and data is not easily re-creatable through better event logs. Still, there may be benefit to applying both these methods in tandem for deeper global insights into system optimization.

Ultimately, these efforts may be applied to further autonomous configuration [SAF07], performance [LCSZ04, BGU+09, HSY05], and workload [YBG+10] analysis and correction. Future research may further reduce the still time-consuming human effort required to make sense of data.

3.5 Summary

In this chapter I have explored the reasons why tracing and analysis is widely considered critical to understanding storage system behaviour. I have also described reasons why performing such studies for the sake of research is rare, including: the cost and difficulty of performance analysis, the challenges of operating at scale, and the many considerations that must be satisfied in order to successfully gather the correct data. Finally, I have explored work that relates to my contributions in this thesis. In that context, I have described some of the significant benefits to tracing and analysis in terms of their ability to address storage system complexity as arises from workloads of increasing intensity and increasingly complex feature sets. In the next chapter I will discuss the implementation of the tools I have developed to perform tracing and analysis, and the ways in which they facilitate and advance the state of storage system studies. Subsequent chapters discuss the specific case studies that demonstrate the benefits of my tracing and analysis framework.

Chapter 4

Data Collection and Analysis

This thesis addresses better understanding of storage system behaviour as a remedy for growing system complexity. One of the challenges organizations face in deploying storage systems is matching the sophisticated array of features and system parameters to the workload patterns they observe. A large part of this problem is that in many cases users lack an accurate characterization of their workloads.

Today there a number of approaches to this problem that are used in practice. For example, an IT department that is concerned with data capacity management can, at some significant investment of time, deploy a commercial deduplication system on a trial basis and measure the potential benefit. This may lead them to understand how that one commercial system might benefit their environment. Similarly a file system designer who is concerned about storage system performance might benchmark against some (usually synthetic) workload. These are useful practices that will no doubt continue to benefit a broad range of enterprise storage environments, particularly those that have a specific concern in mind.

Unfortunately, there are few off-the-shelf tools that help an enterprise or software author to *understand*, in a general sense, the challenges an orga-

nization has around storing and processing data efficiently. They may lead them to be unaware that their workload includes intrinsically inefficient operations, or that simple methods of data capacity reduction they already have available would suffice. As I have argued, deep storage stacks, large data sizes, and I/O intensive workloads obscure what storage systems are doing, and analyzing these aspects of any environment is currently difficult to do in part because of the lack of effective tools for measurement. This shortcoming afflicts researchers and designers as well, who must often invest in building analysis tools before they can retrieve data, and must convince administrators that the custom tools they have developed are safe.

In this chapter I discuss the tools and methodology that I created and used to perform the data collection that I have described in the prior chapter, and the case studies and analysis in the chapters to follow. The tools detailed here are all novel, and were designed to allow an organization or individual to more easily collect and analyze a large and rich trace of storage stack behaviour or file system content. In some cases, such as my investigation of deduplication rates, these tools gather specific information to answer a focused (though common and important) set of questions. In others, the tools gather general purpose data that can be used to characterize file systems and their workloads. This chapter is organized into three primary sections, which each describes the architecture of a component of software that was developed for this thesis.

- **Section 4.1 - The File System Content Scanner**, which I developed to collect file and file system metadata and disk content for the 2009 study of file systems and their contents at Microsoft.
- **Section 4.2 - The Workload Tracer**, which I developed to collect multi-level workload traces at UBC in 2010.
- **Section 4.3 - Data Analysis**, which is the system I developed to analyze and process results from the studies above.

For each tool, I describe its design and implementation, the data model it subscribes to, and as an example, the methodology I used in the case studies where they have been employed. This set of tools is designed to be sufficiently straightforward that it can be deployed by system administrators to better understand their own environments, but just as readily useful to researchers looking to perform studies of live systems.

4.1 File System Content Scanner

The file system scanner is designed to completely scan the file system of a typical desktop workstation with minimal impact to the user of that system. Its measurement capabilities include typical file and file system metadata, and also information about the degree of duplicated content within files.

This is challenging because: file system I/O is relatively expensive and some file systems are quite large, the hashing required to analyze duplication rates is CPU intensive, and different approaches to analyzing duplicate content each require different hash functions on different file system APIs.

This section describes the design of the scanner, and the methodology I put into practice to gather real data from live systems, as it pertains to:

- **Data content** for duplicate content analysis, as will be discussed in Chapter 5.
- **Data fragmentation**, as will be discussed in Chapter 6.
- **File system metadata**, as will be discussed in Chapter 7.

4.1.1 Data Model

The File System Content Scanner records metadata about the file system itself, including age, capacity, and utilization. The scanner reads and records metadata and content records for each file to a log. It reads Windows file metadata [Cor10a], including path, file name and extension, and time stamps. It records any retrieval and allocation pointers, which describe fragmentation and sparseness respectively. It also records information about the whole system, including the computer's hardware information and the time at which the defragmentation tool was last run. It takes care to exclude the pagefile, hibernation file, the scanner itself from its records. A complete list of the data recorded, and the resulting file format, is shown in Appendix B.

In addition to file metadata, my scanner processes file data for the purpose of analyzing duplication, including sub-file granularity duplication across a pool of file systems. The scanner does this by breaking the data in each file into chunks using each of two chunking algorithms (fixed block and Rabin fingerprinting [Rab81]) each with 4 chunk size settings (8KB- 64KB in powers of two) and then computes the hashes of each chunk for each of the

8 possible parameters. One can find whole file duplicates in post-processing by identifying files in which all chunks match. In addition to reading the ordinary contents of files, the scanner also collects a separate set of records using the Win32 BackupRead API [Cor13b], which includes metadata about the file and is more appropriate to store file system backups. My scanner also considers each chunking parameter for this backup read interface, so a total of 16 parameters is considered for each file.

4.1.2 Scanner Architecture

The architecture of the file system scanner is shown in Figure 4.1. Its design is based on the principle that each file should be read just once, and that data content hashes can exploit multi-processor parallelism. The scanner is divided into a File Analyzer, which reads file data and metadata, a number of Processing Threads that create and write file hashes and file metadata according to their unique parameters, and a Windowing Manager that ensures that the file data is buffered appropriately for each Processing Thread.

Ultimately, this design is disk bound for most file systems, because it reads files in metadata-order based on the NTFS Master File Table (MFT). It could be further optimized by consuming the entire MFT and sorting retrieval pointers, and then reading data in linear disk-order, associating blocks with files as it proceeds. However, such an approach is more challenging, particularly in that prior to the creation of this tool there was no reliable data as to the extent to which files would be linear on disk.

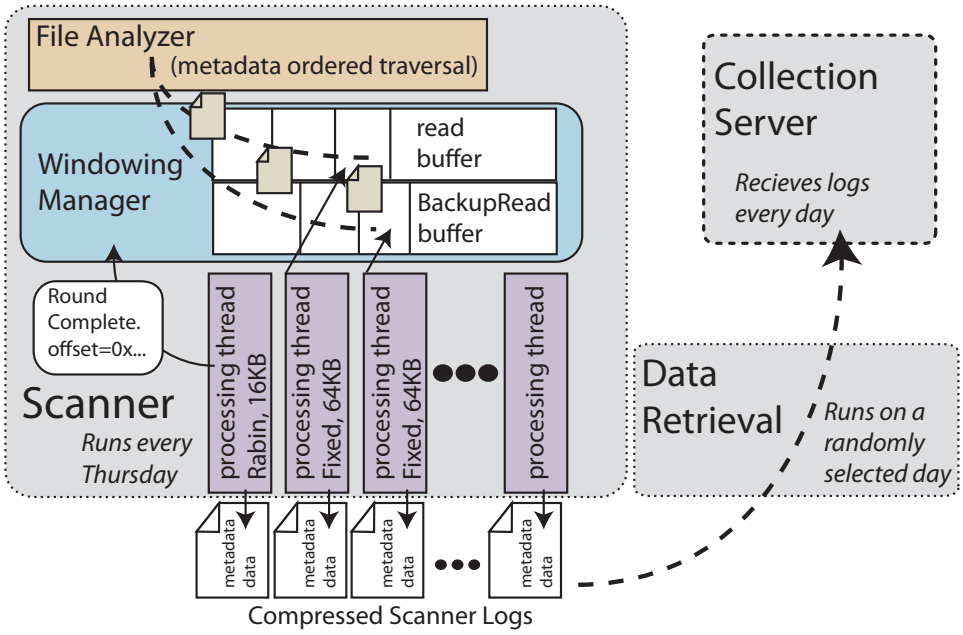

Figure 4.1: File System Content Scanner architecture.

File Analyzer

The scanner includes one File Analyzer that traverses the file system reading files and directories into two buffers, one for the standard read API and one for the BackupRead API [Cor13b]. I assumed that since files are always read together, the two system calls can be serviced from the same Operating System cache. The File Analyzer is free to write to these buffers until each is full, and many small files can fit in each buffer.

Processing Threads

By default the scanner creates 16 Processing Threads which are each assigned a different hash algorithm to explore (8, 16, 32, 64KB hash size) x (fixed or Rabin) x (read API or BackupRead API). These threads each

consume the appropriate buffer (read or BackupRead) and write results independently into a distinct file. Every Processing Thread uses a salted MD5 [Riv92] as the common hash algorithm. When a thread reaches the end of a buffer, it notes the end of the last chunk it completely read, passes that value to the Windowing Manager, and waits to be signalled that its buffer has been refilled.

Windowing Manager

The Windowing Manager waits for all Processing Threads to reach the end of their full buffers. At this point it is the only active thread in the scanner, and a new processing round begins. It takes the minimum progress any thread has made on the last file in the trace and copies the remaining data to the front of the buffer. It then notifies the File Analyzer to continue from where it left off and signals all Processing Threads to wake and continue. This ensures that every thread makes the most progress possible within the buffers, but that the buffers always contain all the data necessary to serve each Processing Thread.

Initialization and Completion

At the start of a scan, the scanner first takes a consistent snapshot of every fixed device (non-removable) file system with the Volume Shadow Copy Service (VSS) [Cor10b]. VSS snapshots are both file system and application consistent. Application consistent snapshots are obtained by hooks that allow any VSS-aware application to save their state cleanly before the snapshot is taken. This allows the scanner to operate on an unchanging

version of the system. It then creates all necessary threads, and files, and collects and writes all file system metadata (e.g., total volume size, machine specification, etc.) to every log.

Scanning of the VSS snapshot proceeds on a predetermined time of the week each week (Thursday at 11pm by default). At the end of each scan, the scanner writes a terminal to each log so I can know that the scan completed fully, and then executes a zip-based compression routine on all log files. Logs are named according to the date so that logs from multiple weeks can be retained without naming conflicts if the Data Retrieval process fails to copy a file fully.

4.1.3 Data Retrieval

At midnight on a night of the week chosen randomly by the installer, the data retrieval process, shown in Figure 4.1 copies the compressed log files to a predefined location. In the case of the 2009 study, this was a file server on the corporate network, and the Data Retrieval process was given appropriate ACLs to create files on the network share, subject to a quota, but not delete or read other files. Scattering collection throughout the week helps smooth the considerable network traffic that is required for a large study. Nevertheless, in a large dataset such as mine, the copying process can result in the loss of some of the scans. Further, because the scanner places the results for each of the 16 parameter settings into separate files and the copying process works at the file level, it is possible to collect results for some, but not all of the Processing Threads.

As part of retrieval, the process also checks for the existence of a specific

kill-bits file on the server named "endofstudy", which indicates to the scanner that the study is complete. When the data retrieval process locates this file, it uninstalls itself entirely. This allows a study author to remotely end the study without user cooperation. Naturally this assumes that users are not adversarial and cannot create a kill-bits file themselves.

4.1.4 Data Collection Methodology

In my 2009 on-disk data study at Microsoft, the file systems under study were selected randomly from Microsoft employees by seeding a pseudo random number generator and producing a list of numbers corresponding to entries in the employee address book. I limited myself to employees located in Redmond, Washington, USA. Each was contacted with an offer to install a file system scanner on their desktop work computer(s) in exchange for a chance to win a weekend retreat for 2 at a nearby resort. In this email I explicitly asked users not to install the scanner on any devices that at any point might not be directly connected to the campus network (e.g., laptops that are taken home) because I could not ensure that such a device could communicate results back to the data retrieval servers reliably. I contacted 10,500 people in this manner to reach the target study size of about 1000 users. This represents a participation rate of roughly 10%, which is smaller than the rates of 22% in similar prior studies [ABDL07, DB99] at Microsoft. Anecdotally, many potential contributors declined explicitly because the scanning process was quite invasive. The scanner ran autonomously in the background starting at 11PM every Thursday between September 18 and October 16, 2009.

4.1.5 Study Biases and Sources of Error

The use of Windows workstations in this study is beneficial in that the results can be compared to those of similar studies [ABDL07, DB99]. However, as in all datasets, this choice may introduce biases towards certain types of activities or data. For example, corporate policies surrounding the use of external software and libraries could have impacted my results.

As discussed above, the data retrieved from machines under observation was large and expensive to generate and so resulted in network timeouts at the data retrieval server or aborted scans on the client side. While I took measures, such a transfers during off-work hours and on random days, to limit these effects, nevertheless it was inevitable that some amount of data never made it to the server, and more had to be discarded as incomplete records. It is likely that, in particular, larger scan files tended to be partially copied more frequently than smaller ones, which may result in a bias in my data where larger file systems are more likely to be excluded. Similarly, scans with a smaller chunk size parameter resulted in larger size scan files and so were lost at a higher rate. In addition, my use of VSS makes it possible for a user to selectively remove some portions of their file system from my study. Among all users, two asked directly how to remove some portions of their file system, and I provided them with that information. I was able to identify one file system where this was apparently the case, as the file system-level space utilization did not match the per-file/directory utilization. This file system was included in the results.

In order to keep file sizes as small as possible, the scanner truncated

the result of every hash of data contents and file names to 48 bits. This reduced the size of the dataset significantly, while introducing a manageability small error factor. For reference, the Rabin-chunked data with an 8KB target chunk size had the largest number of unique hashes, somewhat more than 768MB. I expect that about two thousand of those (0.0003%) are false matches due to the truncated hash.

In addition, during my analysis I discovered a rare concurrency bug in the scanning tool affecting 0.003% of files, in which identical files would be included in the scans multiple times. Although this likely did not affect results, I removed the few files with this artifact.

My scanner is unable to read the contents of Windows system restore points, though it could see the file metadata. I excluded these files from the deduplication analyses, but included them in the metadata analyses.

Finally, due to Microsoft corporate policy, I was unable to extend the prize offer to temporary employees, contract employees or business guests, as only full-time employees are eligible to win prizes as part of participation in internal Microsoft studies of this sort. I opted to make this limitation clear in the invitation to contribute, but to still allow anyone in the address book to opt-in and include the results of scans of their machines. As a results, these employees may have been less likely to contribute to my study. In one case, I am aware that a manager disallowed their 50 reports from contributing, and as a result I revoked their ability to opt-in and eliminated any of their contributions from the study.

4.1.6 Discussion

While file system studies always present challenges, the body of work in my thesis shows that it is tractable for an organization, and that the resulting data is valuable to administrators, researchers, and designers alike. Across nearly 1000 desktop workstations, the results of the entire study is just over 4TB in size (compressed), which can easily fit on two commodity hard drives. Further, as I will show the workload is tractable to analyze even without scale-out resources. The metadata portion of the collected data is extremely valuable in understanding system evolution and is smaller still.

Further, while the disk content analysis of this data (discussed in Section 4.3 of this chapter, as well as Chapter 5) was time consuming, much of the cost was associated with considering many sizes of machine clusters and parameters. Further, much of the processing required machine, as opposed to human time. Finally, the analysis in this paper was performed on single workstations, whereas scale-out approaches could yield much faster answers, potentially even allowing interactive queries.

There are periods of downtime in many enterprise environments [NDT+08] that make this type of scanning convenient as a scheduled process run during off-hours, like defragmentation is now. Deployment could be further simplified by installing a scanner along with other programs pre-installed by IT on all workstations. Annually, or according to some other schedule, results could be gathered from a sampling of available machines.

4.2 Workload Tracer

In addition to my File System Content Scanner, The Workload Tracer is designed to capture high fidelity traces of the file system and block level activity through the stack of a single Windows workstation. It installs as a package of drivers that instrument different layers of the storage stack. This section describes the design of the tracer, and the methodology I put into practice to gather real data from live systems, as it pertains to:

- **Workload characteristics**, as will be discussed in Chapter 6.
- **File system access patterns**, as will be discussed in Chapter 7

An accompanying Linux-based trace replayer was written by a colleague and was used in my evaluation of Capo [SMW+10], which can take file-level traces from this tool and replay them onto a file system for the purpose of system evaluation.

4.2.1 Data Model

The workload tracer gathers metadata about each file system and block-level request. With respect to file system requests, it gathers the size and file offset of the request, and the flags that are issued, which includes information about whether the request can be serviced by the cache or not. It is also capable of richer tracing facilities, including the file name modified as part of the request, and the name of the application that issued the request. At the block level it records absolute disk offset and request size, flags, and in most cases the originating file name. The latest version of the tracer also

includes a hash of the data content itself, for use in analyzing the potential of features like I/O Deduplication [KR10], though these results were not available at the time the system was used to measure live systems in the case study presented in this thesis.

4.2.2 Architecture

The workload tracer is implemented as a paired upper-level filter driver (for block-level requests) with a file system minifilter driver (for file system-level requests). However the two are not strictly isolated. The resulting architecture is shown in Figure 4.2. Requests at the file level are recorded, regardless of their availability in the cache. Cache misses will be reissued from the Windows cache manager and caught in the minifilter driver, with a flag set to read-through the cache, and then eventually seen again by the upper filter driver. Writes to the cache are similarly issued through the filter twice, once to the cache, and again to write-back or -through, depending on the operation.

The upper-level driver also records the name of the process currently issuing the request. Most, but not all, application requests can be correctly tagged in this manner with the calling application. Lower-level requests are not, but the association between the two levels can usually be re-established.

Taking Names

File system requests do not generally include context to directly record file names, so I had to add them. I appended a context pointer to the FileObject structure in Windows and populated this pointer by interposing on all file

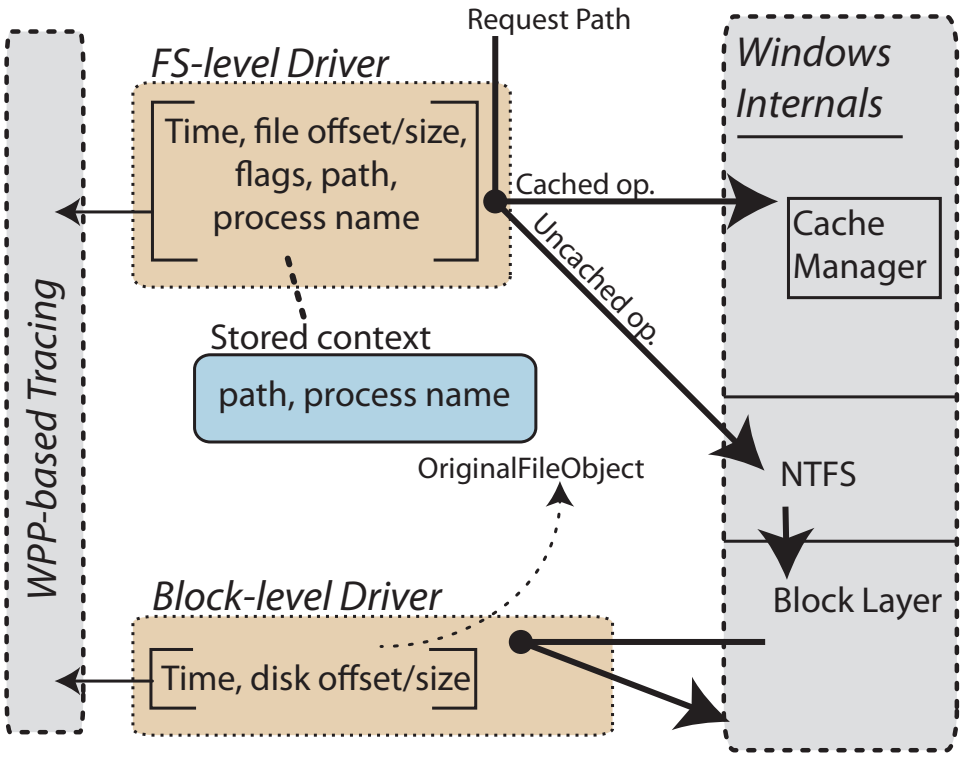

Figure 4.2: Workload Tracer architecture.

creation/open operations, when a complete file name must be present. This method consumes extra memory for the tracer, but avoids potential disk I/O on the read and write paths to look up names. In most cases, the file pointer and associated name can still be retrieved by the lower-level driver through the use of an undocumented `OriginalFileObject` pointer in the Windows I/O Request Packet (IRP) structure. In the 2010 study at UBC, the tracer system was able to make this association successfully in 93% of all trace events. When the file name can't be retrieved safely, the tracer sets the file name to <`UNKNOWN`>.

4.2.3 Data Retrieval

Logs are written as they are collected using the Microsoft Windows Software Trace Preprocessor (WPP). This framework requires a separate process to collect records in real-time, and my tracer installation package includes such a tool, which can be configured to write requests to a file mounted on a remote network share.

Uncompressed and on average, each entry in my dataset consumed less than 60 bytes per request, though the actual space consumed varies depending on the average file path length and the ratio of requests that are serviced by the file and block level of the stack. [4] This is significantly larger than would be possible with a non-rich trace that excluded, say the file that each block request targeted. However, this path information is typically similar between requests which makes it easy to compress, and gzip obtains close to 10x compression over the traces I have gathered.

4.2.4 Methodology

My 2010 study of VDI workstation activity is drawn from a VMware Views installation at the University of British Columbia. In this environment, end users work from Dell FX100 Zero thin clients, while VMs are served from HP BL490c G6 Blades running ESX Server. These servers connect to a Network Appliance 3170s over fiber channel, for booting from the SAN, and 10GigE, for VM disk images. System images are hosted via NFS on a 14 drive RAID group with 2 parity disks. The operating systems and applications

[4] Requests that pass through to the block-level unmerged naturally result in more log entries.

are optimized for the virtual environment [Sch10] and are pre-loaded with the Firefox web browser, Microsoft Office Enterprise, and Sophos Anti-Virus among other software.

Installation of the tracer was significantly aided by the VDI infrastructure itself. In this case the tool was installed only once – directly into the gold master file system image of all systems during the Thursday refresh, which I will describe in more depth in Section 6.1.1. Data was collected to the same centralized storage that served the VDI disk images, which already necessitates a highly reliable high throughput data channel. In this environment, the value proposition to the administrative staff supporting these machines was bolstered by the simple installation, and the insight these tools were able to provide into their otherwise opaque workloads.

Logs from the tracer framework were written to a CIFS network share and collected on the Thursday following a full week of logging. In total I collected 75GB of logs in a compressed binary format. I then checked for corruption, missing logs, or missing events. Out of over 300 million entries I found a single anomalous write to a clearly invalid block address, which I removed. I could find no explanation for the event.

As with the file system scanner, any choice of installation environment necessarily implies some lack of generality, and it is well known that workloads differ in different environments. A fuller comparison of my study to others was provided in Chapter 3.

4.2.5 Discussion

Tracing in this manner does introduce a small performance overhead. Across my synthetic workloads, which I present in Section 7.2, the cost was less than 3% in every case.

In a live deployment of dozens of virtual machines or more, the storage costs necessary (half a GB per machine per day) are not so large that a history of weeks or months cannot be retained. Old logs could be simplified to a basic statistical model for further savings [TKM+12]. This would enable continuous monitoring of per machine performance, as is already available through VMware View's management interface, but one that is extended to support detailed tracing, such that unnecessary I/O consuming processes can be identified and eliminated, and workload inefficiencies can mitigated, as I will describe in Chapter 6.

Storage logs of this type may well have other uses, including virus scanning. Virus scanning can be challenging to perform within virtual machines themselves, because the density of VM environments may make it more likely that a given VM is turned off during a routine scanning schedule, and the risk that "storms" of virtual machines scanning their file systems at the same time will overwhelm shared resources. In contrast, stored access logs could provide an opportunity for out-of-band analysis of file system access behaviour.

4.3 Data Analysis

Whether the subject of a study is tracing, on-disk analysis, or both, investigators often general many compressed flat log files, containing many gigabytes or terabytes of data. I have found that transforming this data into useful findings is best suited to an iterative process that favours quick results and exploration. This section details a work-flow and the optimizations that have made the processing of the datasets tractable.

4.3.1 Interactive Processing

Many data traces in the academic literature are performed as part of answering a specific predefined question, such as: "How would a new system respond to an existing workload?"[NDT+08]. For these purposes, ad-hoc analysis of flat text files may be an expedient way to arrive at an answer. However, when the subject of inquiry is more general, such as: "What are the characteristics of this workload?" or "What opportunities exist to make this workload more efficient?" one may spend considerably more time crafting queries than it would take to execute them.

For this reason I have found it helpful to invest in translating the data into a database supporting a structured query language. Having a higher level language written for the purpose of data processing helps streamline query creation, minimizes bugs, and allows the investigator to focus on questions about the data instead of optimizing log traversal operations. All processing for my 2009 study was done with SQL queries against SQL Server, and my 2010 trace has been similarly imported into mySQL.

I primarily use SELECT WHERE clauses to isolate relationship in the data. Some example queries written by a colleague are included in Appendix A.

4.3.2 Optimizing for Content Hashes

For very large datasets such as these, even bulk database import poses a significant challenge. At the completion of the 2009 file system content study the resulting dataset was more than 16 TB of uncompressed data. This would would have required considerable machine time to import into a database and considerable space to store.

As a novel optimization to make this data tractable for an enterprise, I was able to significantly optimize the performance and capacity by hypothesizing that the bulk of the data would be in unique content hashes of file content. These hashes were critical to my analysis, in that I wanted to determine how much data in files was unique versus the portion that was non-unique, but the actual value of any unique hash (i.e., hashes of content that was not duplicated) was not useful to my analyses. Further, it is unlikely that the absolute value of a unique hash would be interesting to any useful analysis of the data, since the hashes are essentially random bits.

As an optimization, I was able to post process the data to eliminate the costs of storing unique hashes. The novel algorithm that I present here is efficient, because it requires only two linear passes through the entire dataset to eliminate nearly every unique hash. I added this algorithm as a step in my database import tool. During the first of the two passes over the data, my import tool created a pair of 2 GB in-memory Bloom filters [Blo70].

During this pass, the tool inserted each hash into the first bloom filter. If it discovered a value that was already in the Bloom filter, the value was added instead to the second Bloom filter. I then discarded the first Bloom filter.

In the second pass through the logs, the import tool compared each hash to the second Bloom filter only. If the hash value was not found in the second filter, I could be certain that the hash had been seen exactly once and could be omitted from the database. If it was in the second filter, I could conclude that either the hash value had been seen more than once, or that its entry in the filter was a collision. I recorded all of these duplicated hash values to the database, and skipped over any hash seen just once. Thus my algorithm is sound, in that it does not impact the results by rejecting any duplicate hashes. However it is not complete, despite being very effective, in that some non-duplicate hashes may have been added to the database even though they were not useful in the analysis. The inclusion of these hashes did not affect my results, as there was no later processing step that considered these hashes or depending on the invariant that hashes in the database were not unique.

4.3.3 Anonymization and Release

Ideally, traces and scans of file system workload and structure would be widely disseminated. This is the goal of the SNIA IOTTA Repository [Ass14a] and SNIA SSSI Workload I/O Capture Program [Ass14b] which offer public access to a variety of traces of varying quality and size. Unfortunately, the vast majority of these traces are very small. In most organizations there are several logistical barriers to releasing internal data, and many of these

barriers are ill-suited to technological solutions. However, the most serious concern in the enterprise is generally that some proprietary information critical to business operations will leak with the data. I have successfully addressed this problem with an anonymization process, to the satisfaction of Microsoft Corporation, which retains a legal team that is known to be quite large. [5] In fact, some of workstations in this publicly released were located within the legal group itself.

To anonymize trace datasets, regardless of the trace, I am primarily concerned with leaking:

- Any computer, user, or volume name
- Any file name or extension not in widespread use.
- Any directory name not in widespread use.
- The raw content of any file that is not trivially guessable.

Within these limitations it is important to allow as much analysis as possible. By "widespread use" and "trivially guessable" I only mean to exempt files such as ntfs.dll, which are installed as part of any windows operating system installation.

While there are many cryptographically secure hashes that would be easily applied to these fields, such hashes have occasionally been reversed, and even that remote potential threat is sufficient to worry a decision maker whose only concern is security. To overcome this concern I added the additional security of replacing each unique salted MD5 hash in the source data

[5] Microsoft Legal and Corporate Affairs has operations in 57 offices across 40 countries and regions [Cor14a].

with a random serial number permuted by a cryptographically secure hash. I gave different field types (e.g., file name, user name, data content), random values in a different range of serial numbers, such that comparing, for example file name to data content, could never result in meaningful matches. To make some select file extensions useful, I was able to reserve the option to release a mapping for select extensions from their encrypted format to plain text.

This transformation is made more challenging by large traces or content studies that include billions of hashes, because the look-up to determine the unique serial number applied to a particular hash cannot afford a disk seek or network RTT, and is too large to fit in the main memory of most single workstations. This makes the analysis poorly suited even for the database I have described above.

My algorithm for this transformation takes design lessons from map-reduce-style analyses, and avoids disk seeks by taking multiple linear passes over the logs using only in-memory structures. On the largest collection of hashes in the disk content study, this task completes in under a month using only the background processing time of a single desktop workstation with 16GB of RAM, an Intel i7 3.4GHz processor, and 2 commodity hard drives.

First, non-unique hash entries for each domain were extracted from the scanner database and were merged, sorted, and reduced to their respective unique values, and each associated with a unique 59bit pseudo-randomly generated serial number. This required splitting the hashes into 21 pools, each of which could fit into main memory, and using 5 high order bits in the serial number to identify the pool. Then processing tools made 21 passes

over the files, each time transforming only the hashes the appear in one of the respective pools. Although I elected to use a single machine for convenience, it is important to note that the fundamental algorithm not only makes good use of memory, but would be easy to horizontally scale across a cluster in a Map-Reduce or comparable framework.

4.3.4 Discussion

Data analysis is a critical component of a file system trace or study. It demands nearly as much consideration as the data collection itself. A central observation in this thesis is that the data in a study of file system behaviour typically contains more numerous potential results that may have been initially considered. As an investigator gains familiarity with their results, they will discover new things to investigate. A work-flow such as this benefits from investing in support for a rich query interface, and in my experience an SQL database is one well suited approach.

However, large studies that include data content measures, such as mine, have potential performance challenges in SQL databases. I have demonstrated two optimizations for such a dataset. First, I have detailed an algorithm for efficient removal of unique hashes to prune the dataset to a tractable size. Second, I have shown how some useful transformations such as anonymization benefit from map-reduce processing to operate in rounds that avoid disk-seeks.

4.4 Summary

With the tools I have presented in this chapter, it is possible to conduct a large scale study of file system behaviour or content without an extensive background in programming or file systems. This enables an organization to collect, retain, and even share large and extremely detailed traces of system state and behaviour. This framework is usable with a basic understanding of SQL and Windows system administration.

The information these tools make available is not easily found elsewhere. I/O intensive workloads, large data capacities, and rich feature sets have rendered it difficult to define workstation workloads in simple terms. This resulting complexity obscures many potential optimizations and improvements, and hides waste. For organizations trying to better understand their own environments and how to best service their workloads, my tracing and analysis framework provides immediate benefit. Users can determine, for example, the degree of data duplication in their environment, and associate individual processes with I/O operations. In the following three chapters, I will further detail the benefits of rich tracing and analysis in three case studies addressing *capacity management*, *eliminating waste* in workloads, and developing new features to *enhance performance* in existing systems.

Chapter 5

Capacity Management: Deduplication

Even with the significant declines in commodity storage device cost per GB, many organizations have seen dramatic increases in total storage system costs. There is considerable interest in reducing these costs, which has given rise to deduplication techniques, both in the academic community [CAVL09] and as commercial offerings [DDL+11, DGH+09, LEB+09, ZLP08]. However, there is no one widely accepted approach to deduplication that is considered better than others. In fact, as I will show, the parameter space is quite large and the ramifications to end users are significant in terms of performance and capacity savings. Unfortunately, users have little insight into the opportunities provided by different deduplication algorithms, or how those differences will influence the results on their own data.

Initially, the interest in deduplication had centered on its use in "embarrassingly compressible" scenarios, such as regular full backups [BELL09, QD02] or virtual desktops [CAVL09, JM09]. In these cases, exact or near-exact copies of data are repeatedly created and retained. For such cases, there are many methods by which it is relatively easy to reach high lev-

els of space savings, such as using snapshots to represent virtual machine images or virtual-full backups which don't naively copy unmodified data. However, even as such alternatives are widely available, they are not widely explored by deduplication vendors or academic research, which frequently report deduplication rates as naive multipliers (e.g., 20-time capacity savings) on storage efficiency without detailing how workload and dataset selection respectively lead to such results. Despite the interest in deduplication and the range of applications and solutions, there exist few comparisons of the relative benefits of different deduplication approaches and workloads.

However unexplored, the impact of different approaches to deduplication is measureable, with tracing and analysis. This chapter is a case study demonstrating the value of file system analysis as a tool towards more intelligent management of disk capacity than is commonly done today. Specifically I have sought to provide a well-founded measure of duplication rates and compare the efficacy of different parameters and methods of deduplication. This contribution serves to better inform IT professionals as to where different choices in deduplication may yield acceptable results with less system complexity and performance overheads.

I also report on real-world disk capacity and utilization. I provide a dataset that can be used to guide the implementation of efficient storage systems, by focusing on the effectiveness of their solution's balance of real requirements, as opposed to focusing on the highest compression rate possible.

This chapter is divided into 5 sections:

- In **Section 5.1** I provide a very brief *deduplication overview*, describing the techniques involved and discussing the relative overhead of various approaches to deduplication.

- In **Section 5.2** I present a general characterization of the contemporary *capacity and utilization* of files and file systems, as seen through an analysis of my datasets. The results in this section pertain both to file systems generally, and to deduplication specifically.

- In **Section 5.3** I consider the potential for deduplication in *primary storage* against a point-in-time view of a file system or systems. I consider the advantage of different deduplication algorithms and their parameters. I also consider how file system data characteristics such as the size and number of systems under consideration, the file types contained, and the portion of sparse files provide less-complex alternatives to more aggressive deduplication.

- In **Section 5.4** I consider the overall deduplicability of storage systems in a *backup storage* scenario, by analyzing deduplication rates across one month of my dataset. As before I consider how different alternatives to the most costly deduplication algorithm provide comparable results.

- In **Section 5.5** I summarize this chapter and discuss its implications.

5.1 Deduplication Overview

File systems often contain redundant copies of information: identical files or sub-file regions, possibly stored on a single host, on a shared storage cluster, or backed-up to secondary storage. The more data that is included in a system the more potential for such redundancy exists, and the more potential benefit in reducing data sizes.

Deduplicating storage systems eliminate this redundancy in order to reduce the underlying space needed to contain the file systems (or backup images thereof). Deduplication can work at either the sub-file [DGH+09, UAA+10] or whole-file [BCGD00] level. More fine-grained deduplication creates more opportunities for space savings, but necessarily reduces the sequential layout of some files, which may have significant performance impacts and in some cases necessitates complicated techniques to improve performance [ZLP08]. Alternatively, whole-file deduplication is simpler and eliminates file fragmentation concerns, though at the cost of some otherwise reclaimable storage.

Deduplication systems function by identifying distinct chunks of data with identical content. They then store a single copy of the chunk along with metadata about how to reconstruct the original files from the chunks. Chunks may be of a predefined size and alignment, but are more commonly of variable size determined by the content itself.

The canonical algorithm for variable-sized content-defined blocks is Rabin Fingerprints [Rab81]. Briefly, Rabin Fingerprints uses an efficient hash of a sliding window over the data, and declares chunk boundaries when the

low order bits of the hash value are equal to some sentinel. By deciding chunk boundaries based on content, files that contain identical content that is shifted (say because of insertions or deletions) will still result in (some) identical chunks. Rabin-based algorithms are typically configured with a minimum (4KB in my datasets) and maximum (128KB) chunk size, as well as an expected chunk size which is determined by the number of low order bits upon which a boundary is declared.

There are essentially two possible choices of sentinel value. Many proprietary deduplication implementations use a pre-selected random string upon which to declare a boundary. Alternatively, some intentionally select the 0 string, because small sequences of zeros that appears in a file will hash to this zero sentinel, which will result in a boundary as soon as the minimum chunk size is reached. In the analysis in this chapter I have opted for the latter approach for two reasons. First, it provides the most information about the prevalence of zeros in the dataset. Second, because zeros are relatively common, this will provide the most possible advantage to the Rabin-based deduplication, relative to whole-file deduplication. Since one of my goals was to investigate whether whole-file deduplication could reach the performance of Rabin-based methods, this is the conservative choice. This may somewhat impair the ability to compare my deduplication results directly to some other implementations; however, such a comparison would be difficult anyway, because many commercial implementations already use complicated heuristics to favour performance over compression [ZLP08, KDLT04].

Managing the overheads introduced by a deduplication system is challenging. Naively, each chunk's fingerprint needs to be compared to that of

all other chunks. While techniques such as caches and Bloom filters can mitigate overheads, the performance of deduplication systems remains a topic of research interest [KU10]. The I/O system also poses a performance challenge. In addition to the layer of indirection required by deduplication, deduplication has the effect of de-linearizing data placement, which is at odds with many data placement optimizations, particularly on hard-disk based storage where the cost for non-sequential access can be orders of magnitude greater than that of sequential access. Other more established techniques to reduce storage consumption are simpler and have smaller performance impact. Sparse file support exists in many file systems including NTFS [Cor10a] and XFS [WA93] and is relatively simple to implement. In a sparse file a chunk of zeros is stored notationally by marking its existence in the file metadata, removing the need to physically store it. Whole file deduplication systems, such as the Windows SIS facility [BCGD00] operate by finding entire files that are duplicates and replacing them by copy-on-write links. Although SIS does not reduce storage consumption as much as a modern deduplication system, it avoids file allocation concerns and is far less computationally expensive than more exhaustive deduplication.

Because the magnetic disk technology trend is toward reduced per-byte cost with little or no improvement in random access speed, its not clear that trading away sequentiality for space savings makes sense, at least in primary storage. In the next section I will discuss the current state of storage system capacity and utilization as seen through my datasets. This will better contextualize my analysis of deduplication effectiveness in the section that follows.

5.2 The Capacity and Utilization of File Systems

This section provides a general analysis of file systems utilization and capacity, including the amount of space available and consumed. Features such as compression and deduplication depend on an understanding of common data sizes and the typical pressure they apply to disk capacity, and they benefit from specific knowledge of where and how storage is consumed, as I will show. In addition, storage system designs benefit from an understanding of realistic data sizes and consumption. Therefore, this section is intended to stand alone as an independent contribution, in addition to providing specific context for my deduplication analysis.

5.2.1 Raw Capacity

Figure 5.1 shows a cumulative distribution function of the capacities of all file systems in my 2009 study. On the same graph I have plotted results from the similar metadata-only studies performed in 2000 and 2004 respectively, which are collectively the only large scale published studies of the size of deployed file systems.

One can see a significant increase in the range of commonly observed file system sizes and the emergence of a noticeable step function in the capacities. Both of these trends follow from the approximately annual doubling of physical drive capacity. I expect that this file system capacity range will continue to increase, anchored by smaller solid state disks and user-created partitions dedicated to special tasks on the left. This range will continue step wise towards larger shingled magnetic devices [LSAH11] on

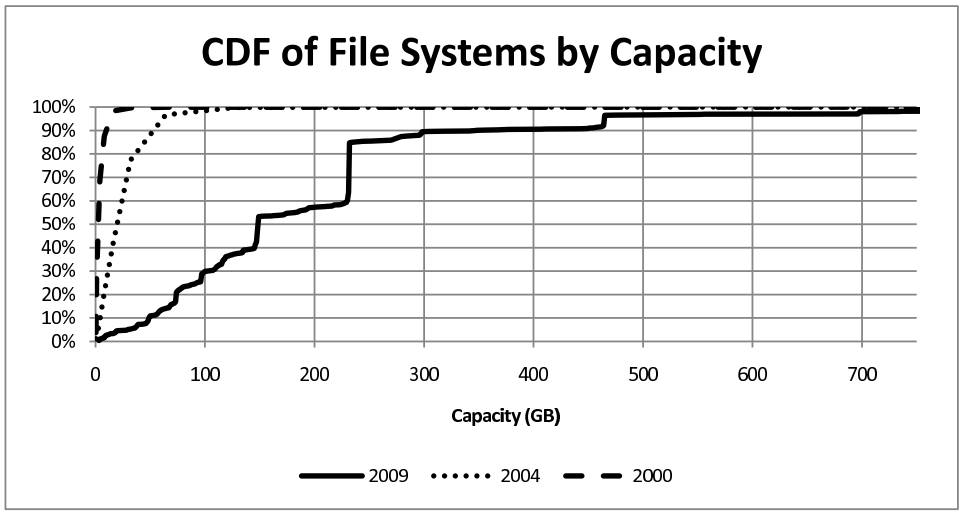

Figure 5.1: CDF of file systems by file system capacity.

the right, which will either force file systems to perform acceptably on an increasingly wide range of media, or push users towards more highly tuned special purpose file systems. The mean file system capacity in 2009 was 194GB, and the median was 149GB, as compared to 32GB in 2004 and 4GB in 2000. This broadening also suggests that research into large scale file systems [KLZ+07, McK03] will have increasing relevance in modern enterprise workstations. This specifically impacts deduplication as well, because deduplication rates increase with the size of the dataset.

5.2.2 File Sizes and Space Consumed

File counts and sizes impact how data for files is best allocated and structured on disk, and can also provide hints at future access patterns. My results show that the overall distribution of file sizes is largely unchanged. Most notably, the median file size in my data set is 4KB, which is the same

in 2004 and 2000, and also true in every other study of file systems I am aware of, going back to at least 1981 [Sat81]).

Figure 5.2 shows the histogram of the occurrences of files of different sizes in 2009, 2004, and 2000. In my 2009 study there is a decrease in the relative occurrence of files between 32B and 256B, and also between 8KB and 64KB. Correspondingly, files between 1KB and 4KB are more common, as are files larger than 512KB.

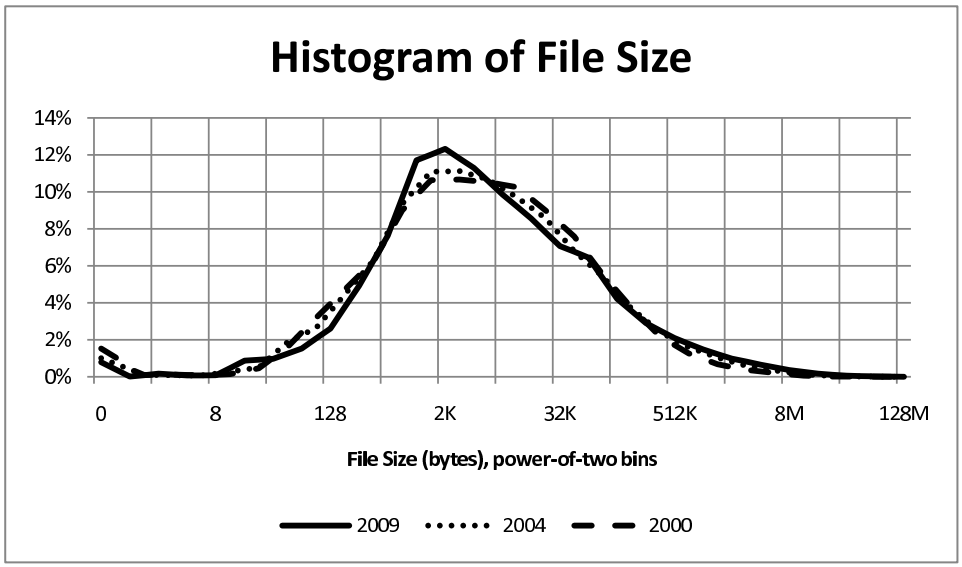

Figure 5.2: Histogram of files by size.

Increasingly, the distribution of file sizes shows a positive skew. Prior work on modelling realistic file sizes utilized a log-normal distribution [AADAD09]. These distribution models were used for accurate benchmarking, and in 2004 showed very low error rates. The results from 2009 suggest that they should remain relevant, provided that a skew term is added and they can be updated with current data. I anticipate that future studies will see this trend towards skewed normal distributions in file sizes exacerbated.

Although the distribution of file sizes shows relative similarity across

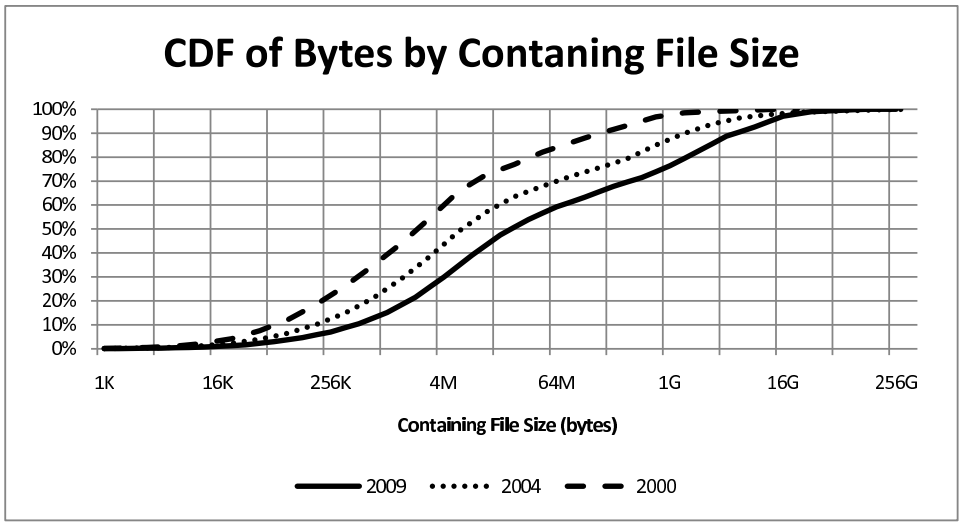

Figure 5.3: CDF of total bytes consumed by containing file size.

years, users do have many more files in their file systems. Next, I consider how the growth in the number of files leads to more dramatic changes in the consumption of space in today's file systems. Figure 5.3 shows a CDF of the total bytes across all files in my study versus the size of file that contains that data. A smooth trend can be observed, as each year the larger portion of data in very large files draws more of the relative consumption compared to smaller files. In 2009, half the data in all file systems are in files smaller than 32MB, versus 8MB in 2004 and 2MB in the year 2000.

A second aspect of the trend towards file system consumption being attributable to very large files can be seen in better detail in Figure 5.4, which plots the histogram of bytes by containing file size. Here, a trend predicted by Aggrawal et al. in 2007 [ABDL07], that a change towards bi-modal file distribution was coming, is well validated by my results. Indeed it seems that bi-modality has continued and is likely to become more pronounced. Further, a third mode above 16GB is now appearing.

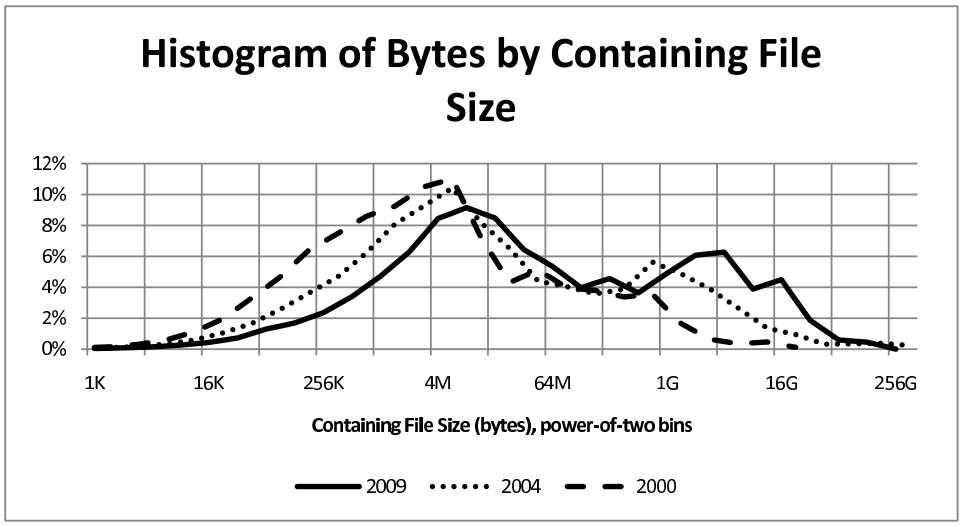

Figure 5.4: Histogram of total bytes consumed by containing file size.

The figure shows that more capacity usage has shifted to the larger files. This is apparent even though there are still few such files in the system because these large files are so very massive – note that Figure 5.4 is logarithmic in the X-axis. This suggests that optimizing for the capacity consumption of large files (if not their workloads) will be increasingly important.

To better place these findings in context, I next consider the type of files in the 2009 study, as shown by the file extension. Figure 5.5 shows the

total bytes consumed by files in my study versus the file extension of files containing those bytes. The ten largest (measured in bytes) file extensions are shown for year 2009, 2004, and 2000.

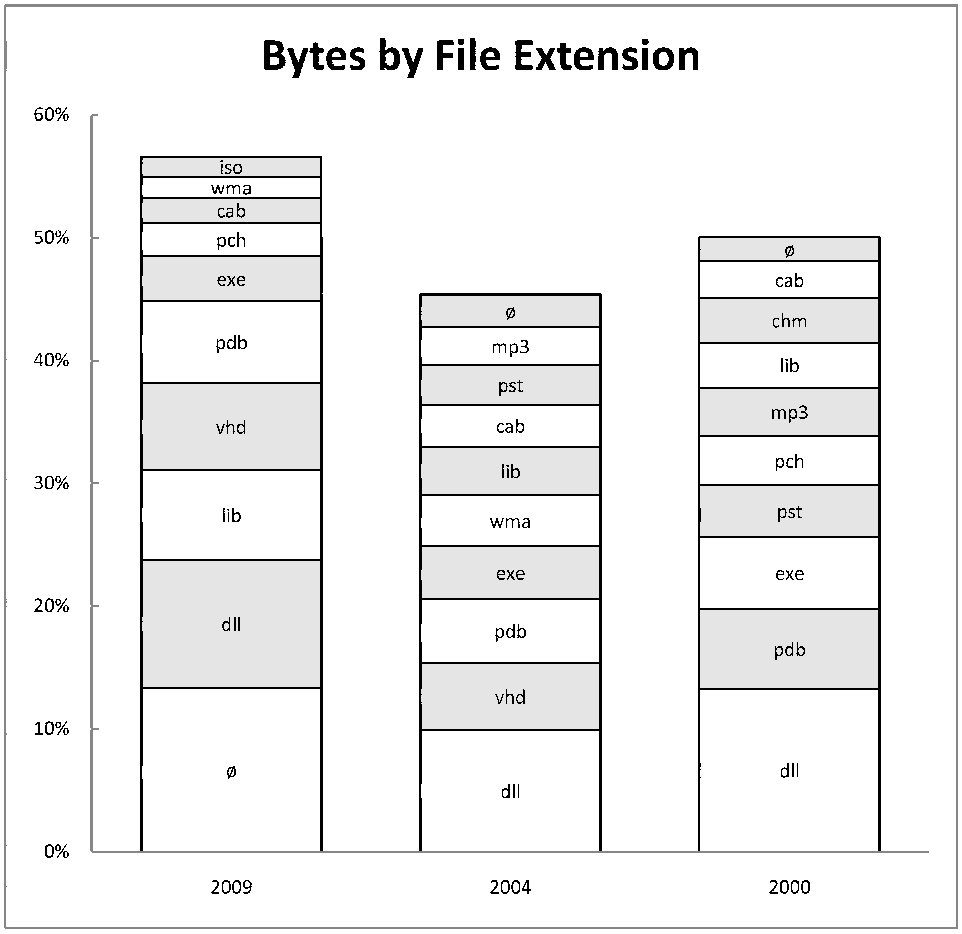

Figure 5.5: Total bytes consumed versus file extension.

Several changes from prior years are apparent in Figure 5.5. Overall, the ten most capacity-hungry file system extensions consume more than 50% of the overall bytes consumed by all files, reversing the trend in prior years towards heterogeneity in file system types, as expressed by bytes. As I will show in Section 7.1.2, the opposite trend can be seen in analyzing file extension in terms of number of files. The portion of storage space consumed by the top extensions has increased by nearly 15% from previous years. Even more dramatically, files with the null extension have moved from the 10th largest consumers of storage to the first. The null extension denotes a file for which no extension is present, as is typical for application-created files that a user is not meant to interact with directly.[6] These files occupy over 10% of the total space consumed in this study. Replacing the null extension at number 10 is .iso files, which are (usually) large image files that are copies of optical media (CD-ROM and DVD). A similar format, VHD files, are a Microsoft image format used for disk drives, usually for virtual machine images. On Windows PCs, this format is common among users of Virtual PC software, a hardware virtualization product. Although VHD has fallen behind .lib (library files) and .dll (dynamically linked library files), owing to the large number of developers in my study, their absolute and relative contribution is higher than previous years.

[6]For privacy reasons, I agreed not to decode full file names and paths in this dataset. My inferences about the NULL file extension is drawn from an ad hoc inspection of several file systems under my own direct control.

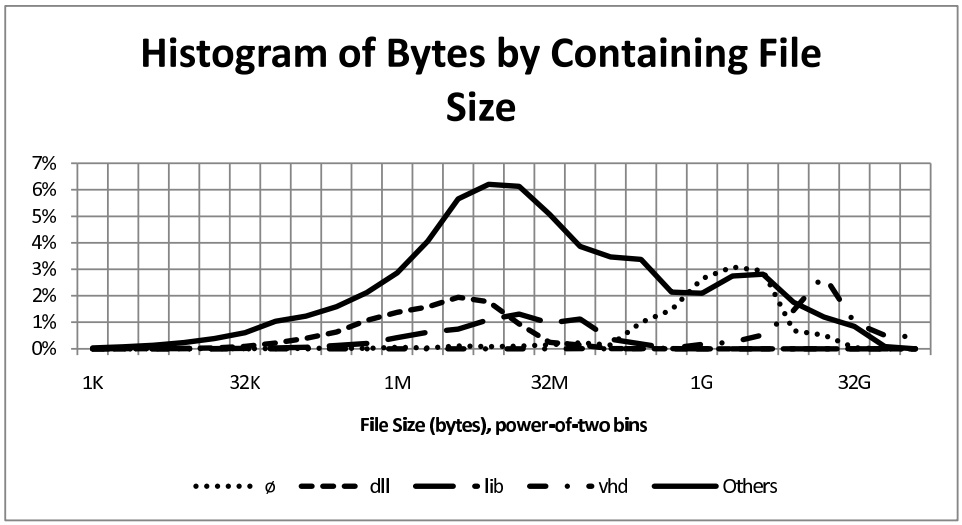

Figure 5.6: Histogram of bytes by containing file extension. Several extensions shown.

These image files, and potentially the null extension files, tend to be very large. To understand the size of these consumers of storage, Figure 5.6 shows a histogram of bytes by containing files, similar to Figure 5.4, but with separate plots for .vhd, .lib, .dll, and the null extension. As expected, vhd files tend to be the largest with a distribution centering around 16GB. Library files are relatively smaller, with distributions primarily between 1MB and 16MB. Less intuitively, the less predictable null extension files tend to be quite large, centering around 3GB. Since null extensions are created by a range of applications (and sometimes users) it is impossible to characterize them simply. However, this result makes clear that the majority of the bytes consumed by files with the null extension are large data repositories, possibly database files.

These opaque image files can be particularly challenging because even

though they appear to the file system as single objects, they are not. Image file formats like VHDs and database files have complex internal structures with difficult to predict access patterns.

Semantic knowledge to exploit complex opaque files, or file system interfaces that explicitly support them may be required to optimize for this class of data. Some systems propose semantically aware optimizations such as file-type specific compression [SPGR08] and workload aware data-layout optimization [YBG+10, emIC+05].

5.2.3 Disk Utilization

Although capacity has increased by nearly two orders of magnitude since 2000, the growth in large files has ensured that utilization has dropped only slightly, as shown in Figure 5.7. Mean utilization is 43%, only somewhat less than the 53% found in 2000. One must assume that this is the result of both users adapting to their available space and hard drive manufacturers tracking the growth in data sizes. The CDF shows a nearly linear relationship, with 50% of users having drives no more than 40% full, 70% at less than 60% utilization, and 90% at less than 80%.

This information is relevant to systems that attempt to take advantage of the unused capacity of file systems. Such mechanisms will be more resilient to the scaling of file system capacities when they assume a constant amount of free space or a scaling of free space proportional to the capacity of the system. This is the case in Borg [BGU+09]) where Bhadkamkar et al. proposed to reorganize on-disk data using a fixed portion of free space. In contrast, in the FS2 project [HHS05], Huang et al. attempted to use

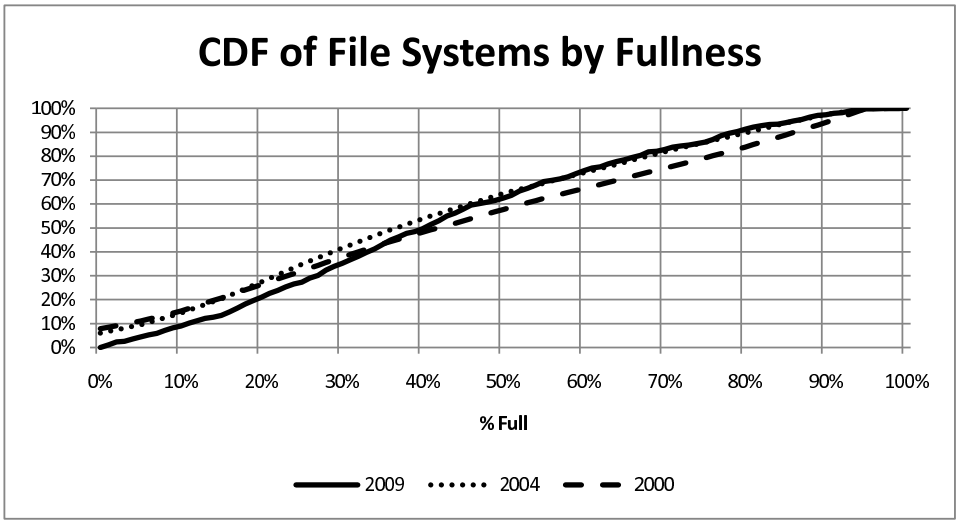

Figure 5.7: CDF of file systems by file system fullness.

free disk space to create linear copies of all common access patterns and persist them to disk in order to improve disk read performance. With this latter approach, the amount of free space required scales with the number of workloads applied to the system. It is not clear if the number of workloads are proportional to the total size of the storage system, or how this value will scale over time. For these reasons, based on the results of this study it is possible that capacity may become a limiting factor for some systems that might otherwise benefit from the FS2 approach.

System designers and programmers also must take care not to ignore the significant contingent (15%) of all users with disks more than 75% full. Full disk conditions are challenging to recover from, frequently untested, and often poorly managed by many applications and operating systems [Fou13, Cor13a].

That capacity has slightly outpaced utilization suggests that capacity

management in large drives is unlikely to be the most valuable optimization, particularly as disks show no signs of significant throughput or latency improvement. However, there still exists significant interest in cost savings through capacity management, and some storage systems favour the use of smaller capacity drives to increase the ratio of throughput and latency to total system capacity. For these reasons, capacity management remains relevant, and so the obvious question is: How might we best manage capacity in a storage system? I address this question in the following sections.

5.3 Deduplication in Primary Storage

In this section, I will measure the efficacy of deduplication in primary storage as shown by my study of file system contents at Microsoft. I chose the week of September 18, 2009 from this dataset to analyze, which means that results were collected on each machine Thursday night of that week. Although there exists a slight variation in the time that scans were started, I treat them all as a single point in time. This dataset includes hashes of on-disk content in both variable and fixed size chunks of varying sizes.

5.3.1 The Deduplication Parameter Space

There are two primary parameters that I can vary in processing this data: the deduplication algorithm/parameters, and the set of file systems (called the deduplication domain) within which duplicates can be found. Duplicates in separate domains are considered to be unique contents.

First, I consider the effects of varying the chunking size of the deduplica-

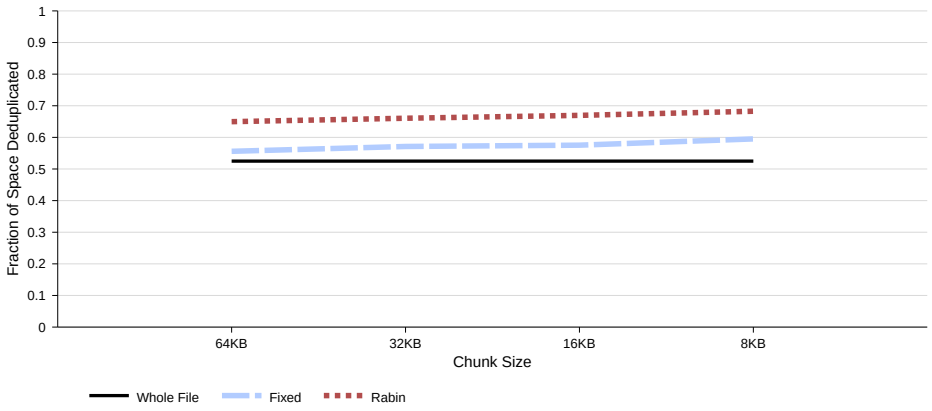

Figure 5.8: Deduplication vs. chunk size. Whole file deduplication, fixed block deduplication (8KB), and Rabin-based deduplication (averaging 8KB) included.

tion algorithm. The size of chunks selected by the deduplication algorithm has been the source of considerable research towards improving the performance of deduplication systems by selecting larger chunks [KU10], and thereby limiting the number of seeks incurred when processing workloads on the system. Figure 5.8 shows the space deduplicated, measured as:

$$1 - \frac{\text{Number unique chunks}}{(\text{Number total chunks})} \tag{5.1}$$

Figure 5.8 plots these results against the average chunk size for fixed-chunk and Rabin-based [Rab81] deduplication algorithms, against whole file deduplication which doesn't depend on chunk size, and so varies only slightly due to differences in the number of zeroes found and due to variations in which file systems scans copied properly; see Section 4.1. This graph assumes that all file systems are in a single deduplication domain; the shape

of the curve is similar for smaller domains, through the space savings are proportionally reduced for all algorithms. As expected, the Rabin algorithm achieves the highest level of deduplication, followed by fixed-chunk, and then whole file deduplication. More interestingly, the difference in moving from 64KB to 8KB chunk sizes is quite small for the chunking algorithms, roughly 5% across the entire range. Depending on the coldness of data and cost of storage per byte, this may be practical. However, I expect that for most primary storage systems the relatively high access frequency and need for low latency would drive users towards the increased performance of the larger chunk sizes, or to whole file deduplication. Also notable is the fact that the different approaches are roughly 10% apart. Moving from the most costly but highest compression 8KB Rabin algorithm to the least costly but lowest compression whole file deduplication yields just under 20% additional space deduplicated.

Next, I consider the effect of varying the deduplication domain size on each of the three deduplication algorithms. Rather than presenting a three dimensional graph varying all parameters, I show a set of slices through the surface, considering the extremes of 8KB and 64KB chunking to show the range of compression achievable.

The set of file systems included corresponds to the size of the file server(s) holding the machines' file systems. A value of 1 indicates deduplication running independently on each desktop machine. Whole Set means that all 857 file systems are stored together in a single deduplication domain. I considered all power-of-two domain sizes between 1 and 857. For domain sizes other than 1 or 857, I had to choose which file systems to include

together into particular domains and which to exclude when the number of file systems didn't divide evenly by the size of the domain. I did this by using a cryptographically secure random number generator. I generated sets for each domain size ten times and report the mean of the ten runs. The standard deviation of the results was less than 2% for each of the data points, so I don't believe that I would have gained much more precision by running more trials. As it was, it took about 8 machine-months to perform these analyses.

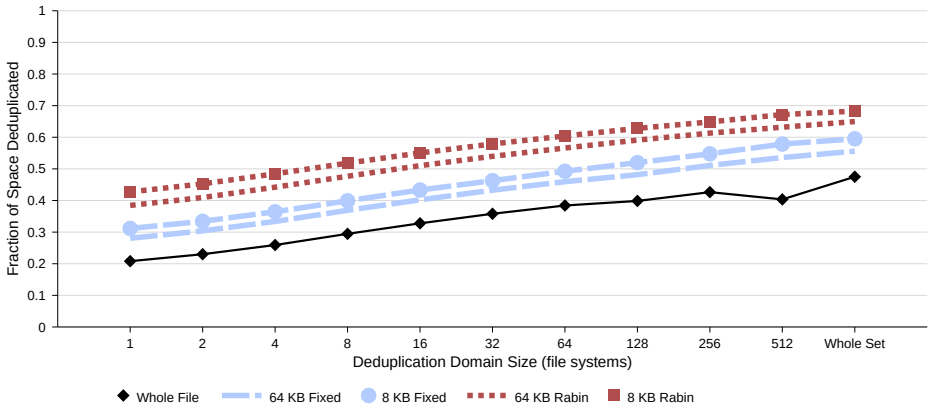

Figure 5.9: Deduplication vs. deduplication domain size.

Figure 5.9 shows the effects of varying deduplication domain size. Here, a much larger range of results can be observed. Single machine performance can vary dramatically from 20% space recovery to nearly 45% depending on the method chosen. However, as the size of the deduplication domain increases, the following trends appear:

First, the space that can be deduplicated by every algorithm increases significantly. The change end-to-end in my dataset is roughly 30%, notably

larger than the difference between the most extreme choices of algorithm in Figure 5.8. Second, the relative advantage of fine-grained deduplication remains roughly constant, at about 20%, regardless of the group size. Finally, as described in Section 5.1, the performance cost of deduplication increases with larger group size, both because there are more total chunks in the system, but also because the size of the index of all chunks must be grown proportionately, which introduces more seeks during the deduplication process itself.

Together, this suggests that in large deduplication domains, the relative value of fine grained deduplication algorithms will be less important than the size of the domain itself, with higher performance approaches such as whole file deduplication representing a more efficient space in the balance between performance and compression. As with algorithm choice, a key deciding factor is the relative hot or coldness of the data and the cost of storage per byte. For some workload and cost, any position in this space may be well justified. However, for primary storage it seems unlikely that low latency and relatively small working sets are a good match for the most aggressive deduplication techniques, if indeed any deduplication is appropriate at all.

5.3.2 A Comparison of Chunking Algorithms

Intuitively, one expects Rabin Fingerprinting to outperform, say, whole file deduplication, because it will catch smaller sub-sequences of bytes that are shared between files. However, these improvements are not necessarily spread evenly across all files.

Extension	8KB Fixed %	Extension	8KB Rabin %
vhd	3.60%	vhd	5.24%
pch	0.56%	lib	1.55%
dll	0.55%	obj	0.84%
pdb	0.44%	pdb	0.56%
lib	0.40%	pch	0.55%
wma	0.33%	iso	0.52%
pst	0.32%	dll	0.51%
<none>	0.29%	avhd	0.46%
avhd	0.27%	wma	0.37%
mp3	0.25%	wim	0.35%
pds	0.23%	zip	0.34%
iso	0.22%	pst	0.34%
Total	7.27%		11.64%

Table 5.1: Non-whole file, non-zero duplicate data as a fraction of file system size by file extension, 8KB fixed and Rabin chunking.

To better explain how specific types of data deduplicate differently, Table 5.1 shows the space deduplicated by 8KB Fixed and 8KB Rabin algorithms versus the extension of the file containing those deduplicated chunks. To understand the context in which chunking algorithms outperform whole file deduplication, the deduplicated space in this table excludes chunks of all zeros, as well as chunks that appear in whole file duplicates. Thus one can see where much of the 20% and 10% improvements in Rabin and fixed-chunk deduplication originate from, respectively. In both cases, VHD files

are a significant contributor. Recall that VHD files are themselves entire file systems containing many small files, stored for the purposes of Operating System Virtualization.

Since these VHD files likely contain Windows operating systems, they share many of the same library and system files as other file system in the deduplication domain, and also are a significant contributor to total storage. Because whole file deduplication does not currently penetrate that opaque type, it is unlikely to find any space reclamation in a pair of VHD files, unless a file is copied and not modified. Cumulatively, the top 12 types for both Fixed and Rabin-based chunking in Table 5.1 comprise more than half the advantage to each of those methods.

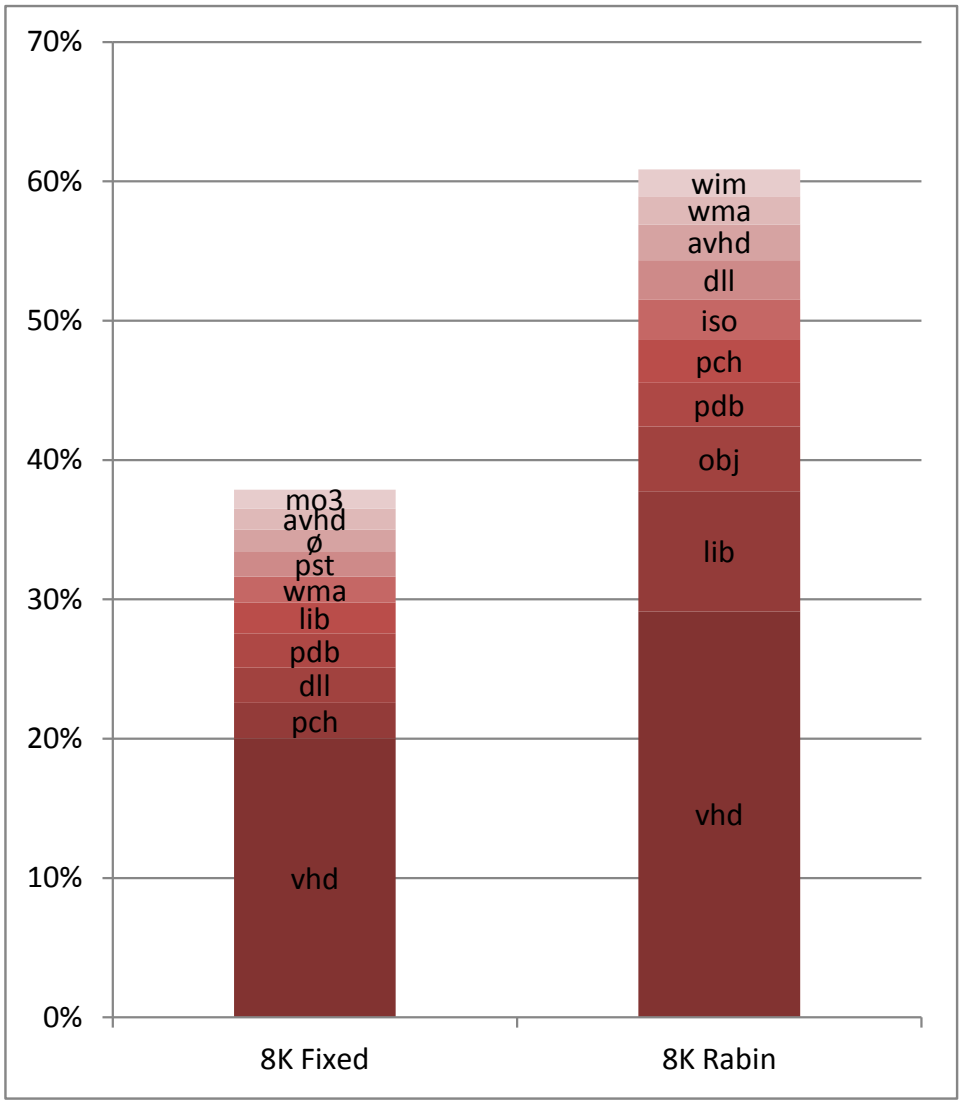

Figure 5.10: Percentage contribution to deduplication by file type. Relative contribution of the top 10 file types to 8KB fixed and 8KB Rabin-based deduplication.

The results above suggest a simple approach to hybridizing deduplication, in which whole file deduplication is used for all but the most well known file types, where a more aggressive deduplication technique can be employed. Figure 5.10 shows the the relative contribution of the 10 types that contribute most to chunk-based deduplication, as a percentage of the total advantage provided by the most aggressive Rabin-based approaches. Thus, if one employed whole file deduplication combined with 8KB fixed chunk deduplication on the 10 types shown in the "8KB Fixed" column of Figure 5.10, they might reclaim as much as 40% of the gap between whole file and 8KB Rabin fingerprinting over the workstations seen in my study.

5.3.3 Characterizing Whole File Chunks

Next, I consider factors that influence the compression rates of whole file deduplication. This is useful to consider because even whole file deduplication carries with it overheads, and in many cases it may be advantageous to be selective in where it is applied. The benefits of whole file deduplication are not evenly distributed. In my data set, half of all file systems achieve approximately 15% savings, but some benefit significantly more, reaching and exceeding the average level of compressability in individual file systems with fine-grained deduplication (roughly 40%).

There is a considerable difference between the average size of a file and the size of file that contributes to the space savings in a whole file deduplicated system. Figure 5.11 shows a skew towards deduplicable bytes being stored in mid-sized files, particularly between 2MB and 128MB, as those are areas where the increase in cumulative bytes in duplicate files outpaces

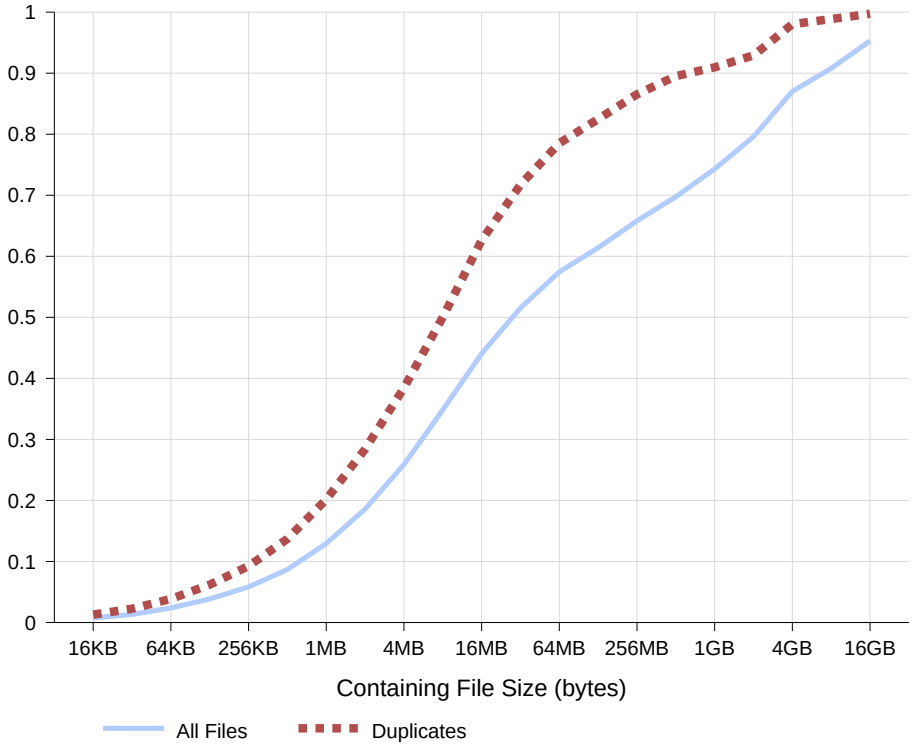

Figure 5.11: Bytes by containing file size. CDF of bytes by containing file size of whole file duplicates and for all files.

that of all files. Not present in this graph are files larger than 16 GB which would appear to the right of the graph area shown, because they are so rarely duplicated. I discuss the presence of very large files in file systems in Chapter 7.

To understand the composition of these mid-sized files, Table 5.2 lists the top 15 file types ranked by the percentage of duplicate space they occupy across all workstations in the study. Mean file size is included in this table to give a feel for the size, but note that the large files of each type will add much

more to the duplicate space than the small files. The DLL and the related lib extension are the most common, together responsible for nearly a third of primary storage duplicate space. In total, the top 15 types alone account for 75% of the duplicate space, while only contributing just over half the total space in the systems measured. More promising is that they together form less than a third of the total files in the system. This discontinuity points to a potential opportunity for a whole file deduplication heuristic that favours the file types most likely to be deduplicated, either at the exclusion of other types, or as a priority optimization. It takes 47 types cumulatively to reach 90% of the whole file duplicate space in the data in my study.

Extension	% of Duplicate Space	% of Total Space	% of Files	Mean File Size (bytes)
dll	20%	10%	6%	521KB
lib	11%	7%	2%	1080KB
pdb	11%	7%	1%	2MB
<none>	7%	13%	5%	277KB
exe	6%	4%	2%	572KB
cab	4%	2%	0%	4MB
msp	3%	2%	0%	15MB
msi	3%	1%	0%	5MB
iso	2%	2%	0%	436MB
<a guid>	1%	1%	0%	604KB
hxs	1%	1%	0%	2MB
xml	1%	1%	5%	49KB
jpg	1%	1%	2%	147KB
wim	1%	1%	0%	16MB
h	1%	1%	8%	23KB
Total	75%	53%	32%	N/A

Table 5.2: Whole file duplicates by extension.

5.3.4 The Impact of Sparse Files

My study includes fine-grained 4KB regions of sparse data – that is, data containing only 0s. Many file systems, including Windows NTFS, the subject of this study, include the ability to represent sparse regions in metadata in order to save space. Somewhat surprisingly, the results of my file system metadata study (see Chapter 7) suggest that this feature is seldom used, occurring in less than 1% of files.

In Table 5.3 I have plotted the total storage system utilization after whole file deduplication in 3 scenarios. First, without any sparse file support, then with aligned 4KB sparse regions, and finally with 4KB sparse regions aligned to Rabin-based boundaries. This simulates what a file system may be able to reclaim with sparse support.

Group Size	Whole File Dedup.	Whole File w/ aligned 4KB Sparse Regions	Whole File w/ Rabin 4KB Sparse Regions
1	79%	77%	77%
2	77%	75%	75%
4	74%	72%	72%
8	71%	69%	68%
16	67%	66%	65%
32	64%	62%	62%
64	62%	60%	60%
128	60%	58%	58%
256	57%	56%	55%
512	60%	58%	57%
Whole Set	52%	51%	50%

Table 5.3: The impact of sparseness on capacity consumption. Utilization is shown as a fraction of raw-data.

The results show that sparseness has a surprisingly small impact beyond whole-file deduplication, usually 1%-2%. This small advantage may

be worthwhile, for example, when data is not frequently re-written. In that case the cost of sparseness is mostly negligible. Further, there are other performance advantages to sparseness. Sparse regions can be more efficient to write to disk because it only requires a metadata update. However, in terms of capacity savings, the benefits of eliminating 4KB sparse regions is very small. Furthermore, allowing unaligned sparse file regions does not appear to be particularly useful in most cases, providing a less than 1% benefit.

5.4 Deduplication in Backup Storage

I now turn from analyzing point in time deduplication rates to considering the deduplication rates of a set of file systems that are being backed up.

In practice, some backup solutions are incremental (or differential), storing deltas between files, while others use full backups which completely replicate the file system. Often, highly reliable backup policies use a mix of both, performing frequent incremental backups, with occasional full backups to limit the potential for loss due to corruption. Most of the published performance measurement of deduplication to date has relied on workloads consisting of daily full backups [KU10, ZLP08]. Certainly these workloads represent the most attractive scenario for deduplication, because the content of the file systems is replicated far more frequently than it is modified. My dataset did not allow us to consider daily backups, so I considered only weekly ones. This is still a very aggressive, if not simply more realistic, backup schedule for a large enterprise dataset [Cor13c, Cor13d]. In practice both full and incremental backups are also expired according to a policy,

which may call for retaining backups for a period ranging from a few days to several years [WDQ+12].

With frequent and persistent full-system backups, the linear growth of historical data sizes will usually out-pace that of the running system. Furthermore, secondary storage is generally less latency sensitive than primary storage, so the reduced sequentiality of a block-level deduplicated store is of lesser concern. However the performance costs of data fragmentation incurred when recovering from a deduplicated backup are increasingly well understood to be a problem [LEB13].

It is worth noting that the most common application of deduplication for backup systems is to deploy an inline *stream-based* deduplication system, in which data is streamed from primary storage to secondary storage at some specified frequency, and is deduplicated in the process. In addition, recall that the backups themselves may be incremental. In this context, the meaning of whole-file deduplication in a backup store is not immediately obvious. I ran the analysis as if the backups were stored as simple backups of the original files in each file system. To copy the file metadata in addition to the data, as would be common in a backup workload, I used the Win32 BackupRead [Cor13b] API. For my purposes, imagine that the backup format finds whole file duplicates and stores pointers to them in the backup file. This would result in a garbage collection problem when files are deleted, but those details are beyond the scope of my study and are likely to be simpler than the analogous mechanism in a block-level deduplicating store.

I considered the 483 file systems for which four continuous weeks of

complete scans were available, starting with September 18, 2009. My backup analysis considers each file system as a separate deduplication domain. I expect that combining multiple backups into larger domains would have a similar effect as seen in primary storage, but I did not run the analysis due to resource constraints.

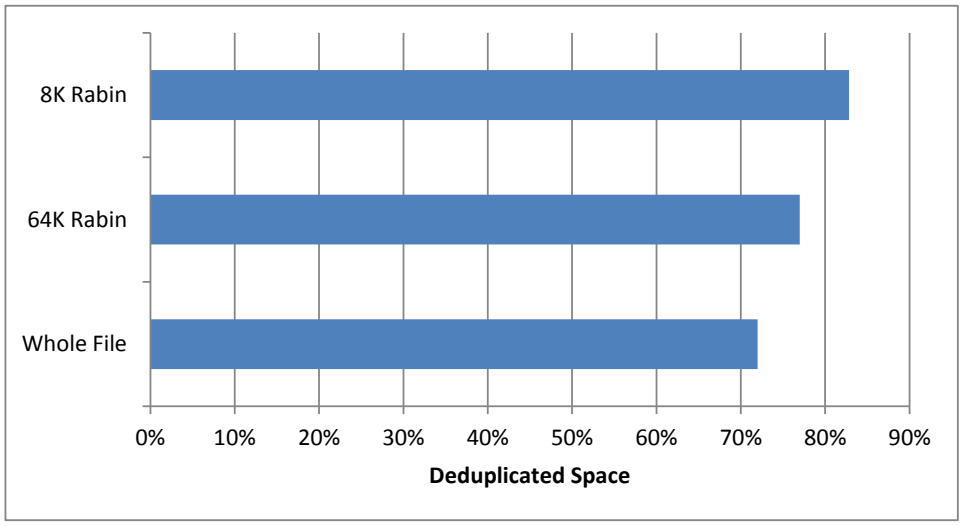

Figure 5.12: Backup deduplication options compared. The comparative space savings of 8KB Rabin, whole file, and 64KB Rabin on a set of backup data.

Using the Rabin chunking algorithm with an 8KB expected chunk size, block-level deduplication reclaimed 83% of the total space. Whole file deduplication, on the other hand, yielded 72%. These results are shown in Figure 5.12, along with results for 64KB Rabin-based deduplication. Necessarily, the 64KB results are not directly comparable, because they are drawn from a different collection of machines – the machines for which I have consistent 64KB data across all weeks. In all cases, these numbers are highly sensitive to the number of weeks of scans used in the study; it's no accident

that the results were around 3/4 of the space being claimed when there were four weeks of backups. In fact, the most significant predictor of deduplication efficiency, by the commonly applied measurements, is the policy under which backup deduplication data is stored. This result was later echoed by Wallace, Douglis, et al. who showed a detailed analysis of deduplication rates by policy and data type [WDQ+12].

In considering Figure 5.12, one should not assume that because 72% of the space was reclaimed by whole file deduplication that only 3% of the bytes were in files that changed. The amount of change was larger than that, but the deduplicator found redundancy within a week as well and the two effects offset.

5.5 Summary and Discussion

The relative merits of deduplication depend on the innate potential to deduplicate a user's data, the frequency of data access, the performance impact of seek requests to access that data, and the cost of storage. My analysis primarily addresses this first concern. For some combinations of those characteristics, for example, when many workstations contain similar data that is very cold, [7] aggressive deduplication will be extremely effective. For other datasets, where data is accessed frequently and performance is important, very simplified or targeted deduplication may be more appropriate. Tracing

[7] Some of my results (not included elsewhere in this thesis for the sake of brevity) provide limited evidence that this may be the case for some minority of enterprise workstation data. More than 30% of files in enterprise workstations are not modified over the lifetime of the file system [MB11]. However, since the Windows operating system does not track last access time, it is not clear how frequently these files are read.

and analysis can provide guidance as to how to employ deduplication in the most effective and lightweight manner to reach an organization's capacity management goals.

Since publishing some of these results in 2010 [MB11], attention has come to the longer term performance implications of aggressive deduplication. Lillibridge et al. found in 2013 that data fragmentation incurred in deduplication can degrade the performance of restoring a deduplicated backup to a problematic degree [LEB13]. Their approach to this problem, like prior efforts to lower the run-time overheads of deduplication [BELL09], is to trade a small measure of compression for higher performance by deduplicating less often or at coarser granularity. My analysis, presented here, shows empirical evidence that these hybrid approaches can be effective, but also shows that simple heuristics based on file type and size may present a simpler approach to deciding when to deduplicate, and what type of deduplication is appropriate. These results demonstrate how analysis of live systems can reveal simple alternatives for efficient and practical solutions to storage management problems.

My dataset also suggests that while sparse files are somewhat ineffective, other simpler deduplication techniques, such as whole file and block deduplication, are relatively good at compressing data. Although each tier of deduplication complexity offers progressively different trade-offs, other aspects of the dataset, such as its size, have a larger impact. I have also found that selectively targeting the files most likely to impact overall compression rates has the potential to further limit the cost of deduplication with a marginal effect on compression rate. These results are directly appli-

cable to the design of space efficient storage systems today, and testify to the ability to use file system analysis to better inform administrators looking to understand their own options for effective management of their respective data.

Chapter 6

Workload Introspection

Storage system administrators face a wide range of concerns about their systems, which may relate to performance, provisioning, utilization, or configuration. In this chapter I propose that while some of these issues are unquestionably important and demand attention, others can be largely understood and addressed with measurement. What administrators lack in many cases is just this – the simple ability to measure and understand what their systems and workloads are doing. With that ability to introspect, solutions to common problems or the elimination of the concern entirely is relatively simple. Put another way, in this chapter I show that there are significant areas of misplaced effort and concern in enterprise workstation workloads, which tracing and analysis can identify and eliminate.

I provide three discrete examples drawn from the measurement of enterprise systems under live deployment. The common thread between each is an administrative concern that is difficult or impossible to satisfy without introspection, the lack of measurement available prior to the publication of my work, and the simplicity with which measurement satisfies the concern or leads obviously to an effective solution.

My first example is performance oriented, and addresses wasted effort in

client workloads, due to multiple clients doing the same work on the same hardware. I present one approach to understand, measure, and eliminate this effort, as was motivated by live system tracing. My second example similarly addresses wasted effort in a workload, but in the form of requests that, although frequent, have no compelling reason to be issued to storage at all. My third example is different in that I use file system analysis to study fragmentation, which is a common concern among system researchers, designers, and administrators. However, in this case, I will show that fragmentation is largely a solved problem and needn't be considered further in most cases. The sections in this chapter each consider one of these examples, as follows:

- In **Section 6.1** I present findings from my investigation into the *performance of a VDI installation* at UBC and describe how, as a case study, tracing was used to inform and evaluate the design of a simple but effective shared cache for VDI storage systems. This cache serves to eliminate waste in the form of identical requests being issued by different clients on the same hardware.

- In **Section 6.2** I describe how tracing has helped to identify a number of instances of *wasted effort* in client workloads, both in a research setting and in a live deployment. These elements of waste in workloads are strictly unnecessary and can in most cases simply be turned off for the benefit of the storage system.

- In **Section 6.3** I present analysis of *on-disk fragmentation* which argues that disk fragmentation, as a concern for storage systems, is largely solved.

- In **Section 6.4** I summarize and discuss the contents of this chapter.

The first two of these examples focus on on enterprise workstations deployed though a Virtual Desktop Infrastructure (VDI), as was used in my UBC trace. Although there are many different storage workloads one could consider, I chose this environment specifically because I had a strong partner in UBC, the VDI space is relatively under-examined in the academic literature, and the sample size is comparable to most published studies of file system workload. VDI systems are also interesting because they include a storage stack that is particularly deep and complex, which presents many opportunities for unnoticed inefficiencies to hide. The final example draws from relevant results in my 2009 study at Microsoft.

In this chapter, my application of tracing and analysis demonstrates that in the complexity of a real workload and cluster storage architecture, important aspects of the system's behaviour are obscured. Leveraging measurement tools to elucidate those characteristics in each case yields opportunities to either deploy a simple solution, or to entirely eliminate an area of concern. I begin by discussing and measuring the VDI environment.

6.1 Performance Analysis of VDI Workloads

Virtual desktops represent the latest round in a decades-long oscillation between thin- and thick-client computing models. VDI systems have emerged as a means of serving desktop computers from central, virtualized hardware and are being touted as a new compromise in a history of largely unsuccessful attempts to migrate desktop users onto thin clients. The approach does

provide a number of new benefits. Giving users private virtual machines preserves their ability to customize their environment and interact with the system as they would a normal desktop computer. From the administration perspective, consolidating VMs onto central compute resources has the potential to reduce power consumption, allow location-transparent access, better protect private data, and ease software upgrades and maintenance.

However, the consolidation of users onto fewer hardware resources in VDI deployments puts enormous pressure on storage because it consolidates sources of I/O load. This section provides both technical background and a performance analysis of VDI environments. From this basis, I present an empirical argument for a a host-side cache called Capo, which was published in 2010 [SMW+10]. Capo is a simple but effective solution to mitigate these pressures. I will begin by describing the environment that Capo operates in. This overview is important because virtualized storage in enterprise environments both enables and obscures many of the insights that tracing and analysis can provide.

6.1.1 VDI Overview

Today, the two major vendors of VDI systems, Citrix and VMWare, individually describe numerous case studies of active virtual desktop deployments of over 10,000 users. From a storage perspective, VDI systems have faced immediate challenges around space overheads and the ability to deploy and upgrade desktops over time. I will now briefly describe how these problems are typically solved in existing architectures, as illustrated in Figure 6.1.

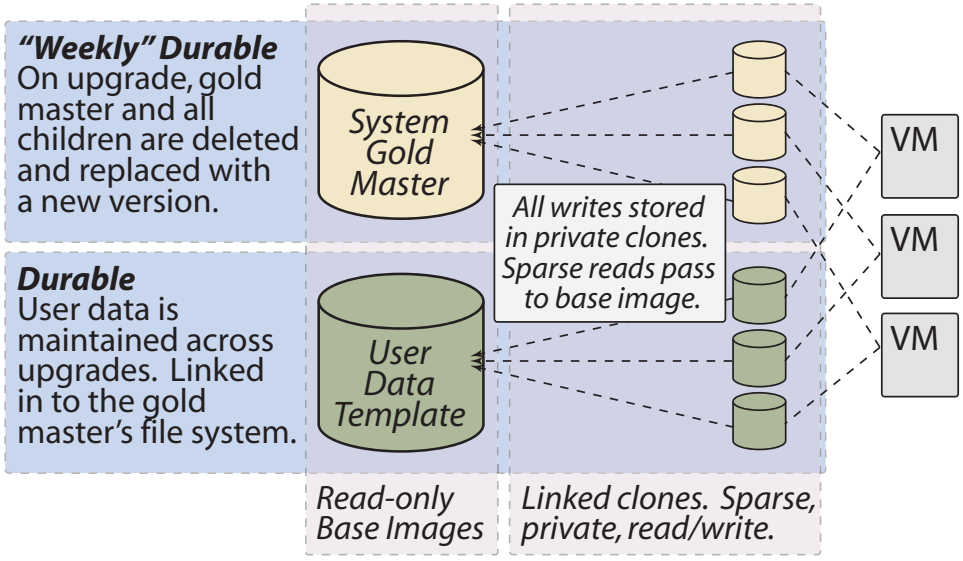

Figure 6.1: Typical image management in VDI systems.

Copy-on-Write and Linked Clones

VDI deployments are organized around the storage of operating system images, which are each entire virtual disks, often tens of gigabytes in size. A naive approach to supporting hundreds or thousands of virtual machines results in two immediate storage scalability problems. First, VMs must have isolated disk images, but maintaining individual copies of every single disk is impractical and consumes an enormous amount of space. Second, to preserve the major administrative advantage of VDI – very flexible deployment, adding desktops requires that images can be quickly duplicated without taking the time to perform a complete copy.

Storage systems today have usually addressed this problem in the form of using *Copy-on-Write* (CoW) files to represent disk images [MAC+08]. As such, this observation is not new, and it has been a recurring challenge in

virtualization. Existing VDI systems make use of VM-specific file formats such as Microsoft's VHD [Mic09] and VMware's VMDK [VMW10]. Both allow a sparse overlay image to be "chained" to a read-only base image (or gold master). As shown in Figure 6.1, modifications are written to private, per-VM overlays, and any data not in the overlay is read from the base image. In this manner, large numbers of virtual disks may share a single gold master. This approach consolidates common initial image data, and new images may be quickly cloned from a single gold master.

Image Updates and Periodic Rollback

Image chaining saves space and allows new images to be cloned from a gold master almost instantaneously. It is not a panacea though. Chained images immediately begin to diverge from the master version as VMs issue writes to them. One immediate problem with this divergence is the consumption of independent extra storage on a per-image basis. This divergence problem for storage consumption may be addressed through the use of data deduplication, as discussed in Chapter 5.

For VDI, wasted storage is not the most pressing concern: block-level chaining means that patches and upgrades cannot be applied to the base image in a manner that merges and reconciles with the diverged clones. This means the ability to deploy new software or upgrades to a large number of VMs, which *was* initially provided from the single gold master is immediately lost.

The leading VDI offerings all solve this problem in a very similar way: They disallow users from persisting long term changes to the system image.

When gold master images are first created and clones are deployed, the VDI system arranges images to isolate private user data (documents, settings, etc.) on separate storage from the system disk itself. As suggested initially in the Collective project [CZSL05], this approach allows a new gold master with updated software to be prepared and deployed to VMs simply by replacing the gold master, creating new (empty) clones, and throwing away the old version of the system disk along with all changes. This approach effectively "freshens" the underlying system image of all users periodically and ensures that all users are using a similar well-configured desktop. For the most part, it also means that users are unable to install additional long-lived software within VDI images without support from administrators.

Discussion

Storage for VDI systems presents a number of interesting new solutions while also creating new problems. For system efficiency, the depth of the virtualized storage stack creates two immediate obstacles. First, administrators can not easily see what client systems are doing because of the opaque virtualization layers. Further, the depth of the stack means that assumptions made at one layer (e.g., the linearity of the underlying address space) are less likely to be correct at lower levels. These observations together suggest that there will be many opportunities for inefficiencies to remain unnoticed. In the remainder of this section (and the next as well) I will use detailed tracing at the client-level to elucidate some of these inefficiencies.

6.1.2 Temporal Access Patterns - Peaks and Valleys

Having explained the structure of virtualized environments as they pertain to storage, I next turn my attention to a specific analysis of VDI system performance. I will explore where significant load comes from in a VDI system, detail the nature and magnitude of the load and discuss how it can be mitigated. First I consider when the load occurs.

In Figure 6.2 I present the access patterns in IOPS across different times of day for a complete week of trace. I have plotted two series on each day. The first tracks peak workloads, which, for the sake of clarity requires some careful selection of processing parameters. First, to preserve the peaks in the workload, without adding so much noise as to make the graph unreadable, I processed the workload to calculate the average IOPS within each 10 second period. IOPS were measured as was submitted to central storage by the clients, and as was measured by the client's own clocks. Next, to eliminate the remaining noise in the graph while emphasizing the peaks in the workload, I have further taken the maximum of each of these 10 seconds periods every 5 minutes to plot. I chose this transformation to strike a fair balance between depicting the periods of high peak burst, without creating so much jitter in the graph so as to make it unreadable without smoothing via an average, or eliminating valleys via taking the maximum across too-coarse time-scales. Had I chosen a smaller averaging window than 10 seconds, one would see much higher peaks, because even within 10 second windows the workload is notably bursty. As such, the absolute value of IOPS in this graph shown should be regarded as being of a lesser fidelity

and thus importance than the relative size and location of peaks and valleys. To provide an additional, less-complicated view of the workload, I have also included a simple average of IOPS over 5 minute time scales in the figure.

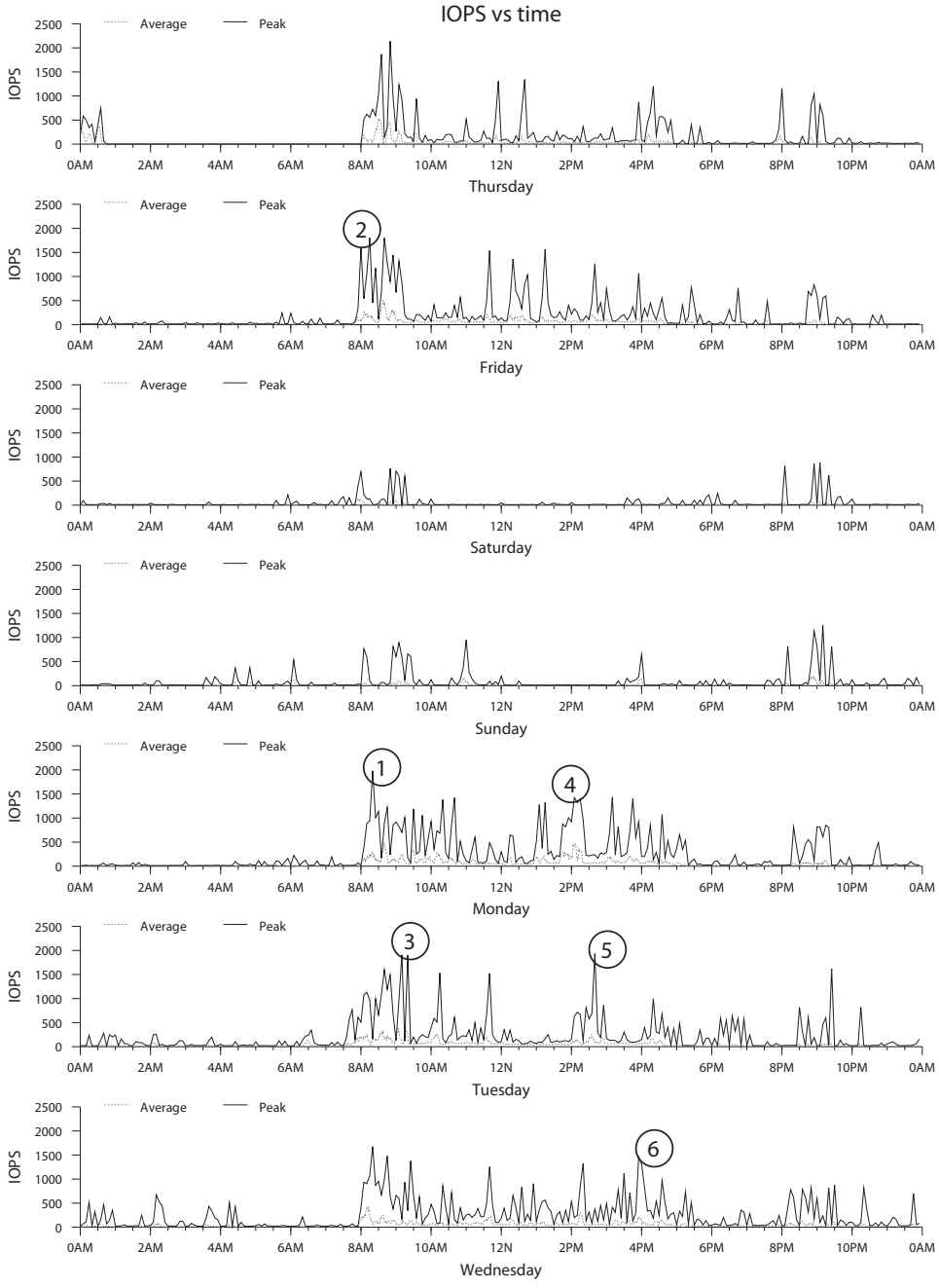

Figure 6.2: Activity measured in I/O operations over each of the 7 days.

Viewed this way, the diurnal aspects of the workload are clear. There is a distinct peak load period between 8:30 and 9:30 every weekday morning, as employees arrive at work. Lunch, dinner, and late nights are periods of relative inactivity, as are the weekends. Late afternoon peaks are sporadic and difficult to predict, but reach loads nearly as high as in the morning.

These peaks are large. Peaks are frequently twice as large as the average IOPS over the range of time in which the peak occurs, which poses a significant problem for storage administrators. If they purchase centralized storage to serve the average or median request load, their users will frequently face periods where load overwhelms storage-capacity by a factor of more than 2:1. Instead, many storage administrators will choose to do what has been done by the administrators of the system studied here - overprovision storage (most of the time) so that peak loads can be served at an acceptable level. Thus the absolute peaks in loads such as these is an interesting problem.

For the purpose of illustration, I have selected 6 interesting peak loads from the trace, and numbered those locations on Figure 6.2. The selection was chosen so as to capture periods where the load is quite high even compared to other peaks, and to sample several different days and times of day in order to capture a breadth of behaviour. Those workloads are explicitly listed in Table 6.1 and will be referenced in the following sections, which will focus on the composition of these peak loads periods and how to efficiently service them.

	Time Period
1	Mon. 8:30am-8:45am
2	Fri. 8:30am-9:00am
3	Tue. 9:15am-9:30am
4	Mon. 2:00pm-2:30pm
5	Tue. 2:40pm-3:00pm
6	Wed. 4:00pm-4:15pm

Table 6.1: Workload peaks selected for more detailed analysis.

6.1.3 VM Contributors to Load

In considering how to best service peak loads, I will first analyze the composition of those peaks in terms of the individual VMs from which the workload is generated. Recall that my VDI-based workload is comprised of a number of virtual desktops issuing requests to a central storage system.

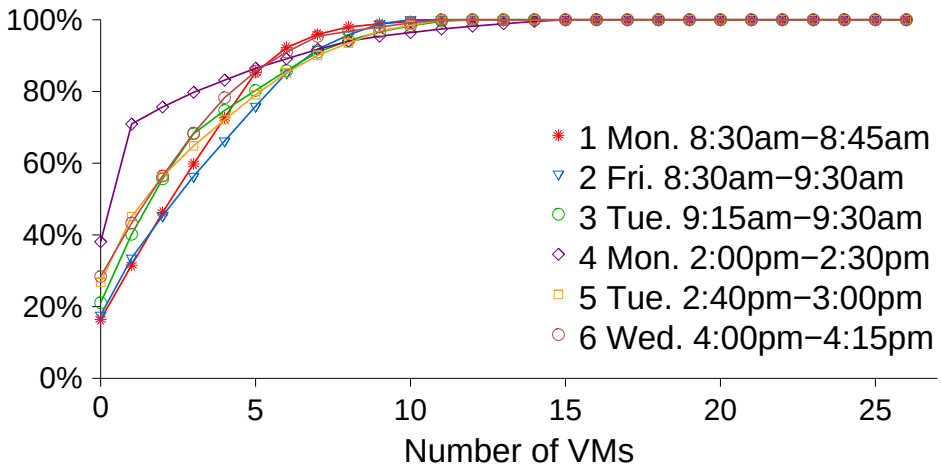

Figure 6.3: Contributors to each of the major workload peaks.

Figure 6.3 shows a CDF of these VM desktops by their contribution to the total workload for each peak, which tells us if peaks are caused by

multiple VMs generating workload equitably, or if peaks are cause by a few outliers during periods of heavy use. In most cases, it is the former; however, the peak in slice 4 was caused primarily by just 4 VMs.

To view this result in more relative terms, I took the 28 virtual desktops which are in regular day-to-day use and calculated the percentage of VMs which contributed at least 5% of the peak workload. As listed in Table 6.1, periods 1,2,3, and 5, and 6 each showed 26%-29% of VMs contributing more than 5% of the total load in their respective periods. From this I conclude that in most cases peak loads tend to be caused by many VMs each experiencing storage-oriented workload at the same time, as opposed to a small number of VMs inequitably creating load. [8] This observation is generally interesting in the characterization of peak loads in VDI systems, and I will leverage it specifically in Section 6.1.5 where I discuss the use of a shared cache in mitigating peak loads.

6.1.4 Disk Divergence

Since it is clear that multiple clients are contributing to peak load periods at the same time, it is natural to wonder if the VMs are engaged in similar behaviour. Since VMs typically use disk images chained from a gold master, one way to consider this question is to measure the rate at which the overlay image diverges from the original image. If divergence occurs quickly and completely then there may ultimately be little similarity between the many VMs contributing to a workload.

[8] This finding would be further magnified in a larger VDI deployment, where the relative load on storage is much higher relative to the power of each individual workstation, and so individual desktops would simply unable to saturate a network resource.

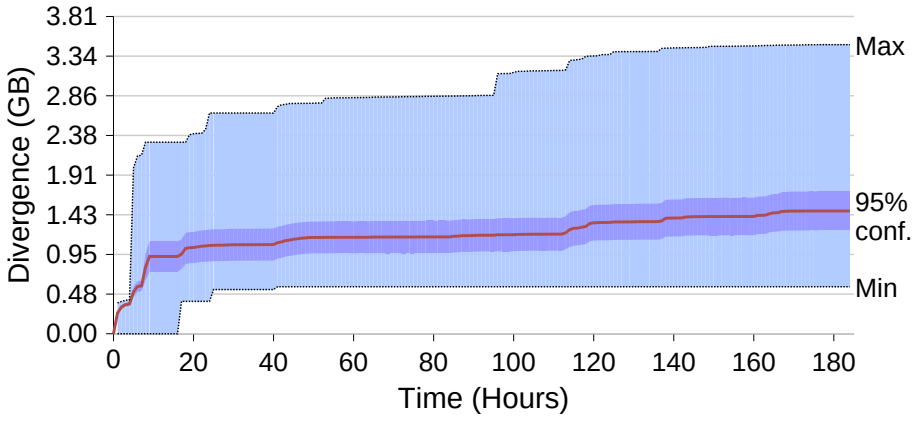

Figure 6.4: Bytes of disk diverging from the gold master.

Based on this question, I calculated the frequency at which the trace observed the first write to a sector. Figure 6.4 plots this data for the average VM, as well as the most and least divergent VM, over the entire study. Within 24 hours, most VMs hit a near plateau in their divergence, around 1GB. Over time this does increase, but slowly. A smaller set of VMs do diverge more quickly and significantly, but they are far from the 95% confidence interval. I conclude that there is usually significant shared data between VMs, even after several days of potential divergence.

6.1.5 Capo: Caching to Alleviate Peak Workloads

Based on the results above, my colleagues and I have developed a prototype client-side cache for virtual machine environments. The system is called Capo and has been analyzed, evaluated, and published [SMW+10] based on my 2010 UBC trace, and the resulting insights about the degree of duplication in VDI workloads.

Note that one could perform the analysis I present in this chapter with tracing at typical fidelity (i.e. tracing individual requests and their block offsets). What distinguishes the measurement in this section is the analysis that considers the impact of the virtual environment on disk divergence and the CoW structure of virtual disks. I am aware of no other analytical research that considers measurement of live systems with these considerations. In both the next section and the following chapter, I consider the advantages of considering extended details available only in my tracing framework.

Capo Architecture

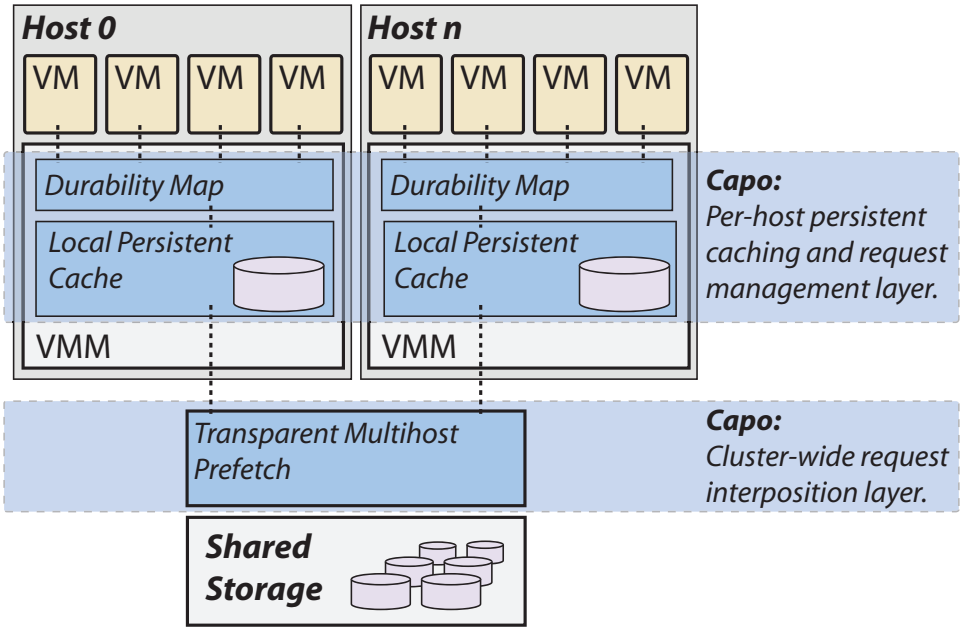

Figure 6.5: The architecture of Capo, a cache for VDI servers.

Capo is a client-side cache for a virtualized server that hosts some or many virtual machines that share gold-master images. The cache is designed to eliminate redundant reads and writes, and Capo also features a broadcast protocol used to further increase sharing by pre-loading the cache with accesses from other servers. Designing a cache on the client side of the storage system removes scalability concerns with caching at the central storage system directly, while retaining the benefits of potential sharing between VMs, as suggested by my trace analysis. The Capo architecture is shown in Figure 6.5. For brevity, I omit a more comprehensive description of Capo, including the multi-host pre-fetching, and a description of the mechanisms included in its implementation. A more comprehensive description of Capo is available in the published paper describing that system [SMW+10].

Based on the finding that peak workloads are distributed among a large number of VMs, I sought to evaluate the potential savings using Capo by determining how much similarity there is between the workload of each individual VM. Intuitively, there may be significant potential to eliminate redundancy if, for example, during initial user login each desktop is accessing the same login-related libraries and login services from storage.

Table 6.2 shows each of the highlighted periods. In the third column, for each period I list the read IOPS and read throughput as a percentage of the total IOPS and total throughput (in bytes) respectively. Like other enterprise workloads in the literature [NDT+08], the VDI workload in these periods shows approximately 2:1 preference for writes, as measured in IOPS, and a 2:1 preference for reads, as measured in throughput. From this I conclude that both read and write requests will ultimately be important to

	Time Period	Read % (IOPS / Thpt)	Duplicate VM Reads	Duplicate Cluster Reads
1	Mon. 8:30am-8:45am	50% / 78%	81%	91%
2	Fri. 8:30am-9:00am	48% / 78%	88%	97%
3	Tue. 9:15am-9:30am	36% / 57%	78%	99%
4	Mon. 2:00pm-2:30pm	38% / 59%	59%	99%
5	Tue. 2:40pm-3:00pm	31% / 48%	77%	97%
6	Wed. 4:00pm-4:15pm	40% / 63%	99%	>99%

Table 6.2: Read request IOPS and throughput, and the percentage of reads that are identical across VMs and across the cluster for each of the peak load periods.

consider in evaluating the potential benefits of a shared cache. However, for the purposes of this discussion, I first consider read requests.

Caching Read Requests

In the 4th column I present the percentage of reads to the same range of bytes that have been previously seen by that VM over the course of the trace. With a large enough cache, one could potentially absorb *all* these reads. However this is not practical, as VDI deployments must scale to a large number of VMs and cannot devote a large cache to each one if they wish to benefit from economies of scale. However, it is promising that these values are so high, particularly because the trace results shown here are downstream of the page cache, which itself absorbs many of the duplicate requests.

The right-hand column presents the same measure, but imagines that caching could be shared across all VMs in the cluster. This is practical to consider because VMs share a gold master image for much of their content, and so reads of a virtual disk (as are shown here) can be known to be identical

when accessing a range of bytes that have been previously viewed by another VM, provided they have not been modified previously by either VM. This finding shows very strong support for the notion of a sharing a cache among different virtual machines. Among the selected peak periods, slice 4 stands out for having an unusually low duplicate read rate for VMs, but a very high rate across the cluster as a whole. I investigated and found that two very active VMs had duplicate read rates of 26% and 30%. By including the most beneficial 62%, 85% and 96% of VMs, I could reach duplicate read rates of 40%, 60% and 90% respectively. From this I conclude that even as you can achieve significant reduction in read requests with caching, possibly even by sharing caches, that some benefits may require careful selection of the VMs in question.

Caching Write Requests

Now I consider the potential benefit of a cache with respect to write requests, which constitute the majority of individual operations. Figure 6.6 shows the percentage of disk writes that overwrite recently written data, for time intervals ranging from 10 seconds to a whole day. I have included results from each of the seven days to underscore how consistent the results are. Friday stands out somewhat as having a lower overwrite percentage, but it is also the case that overall write load on Friday was lower than any other day.

In a short time span, just 10 seconds, 8% of bytes that are written are written again. This rate increases to 20%-30% in 10 minute periods and ranges between 50%-55% for twenty-four hour periods. From this I conclude

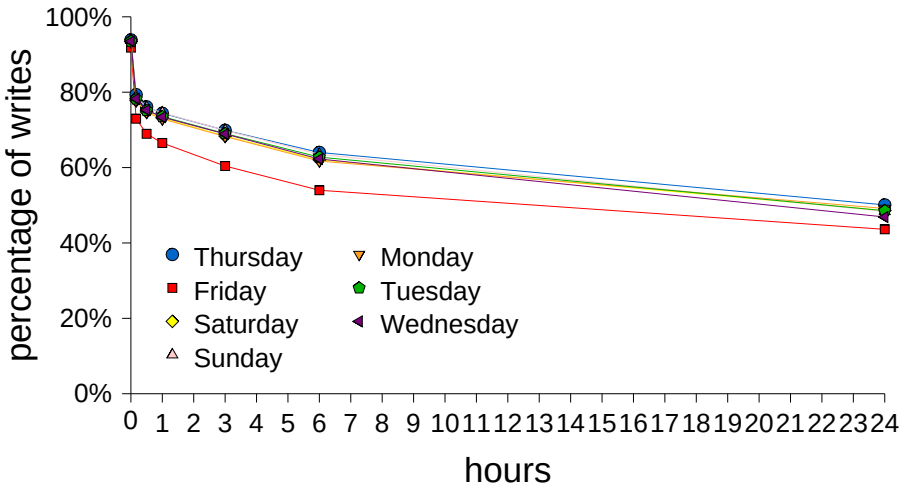

Figure 6.6: The benefits of delaying writeback. The percentage of bytes that need to be written to the server if writes are held back for different time periods. This is lower than the original volume of writes due to the elimination of rewrites.

that considerable system-wide effort is spent on writing data with a high modification rate. This bears some similarity to the results of a file system study of NTFS from 10 years prior conducted by Vogels [Vog99], who found that most files are short lived, and at the time recommended more extensive use the *temporary file* flag. It is worth noting that my analysis suggests that the flag is still not being used extensively in either the live UBC trace or (less surprisingly, because the file system should avoid writing temporary files to disk) the Microsoft on-disk data trace. This result suggests that delaying write-back may be an extremely effective means of lowering peak loads.

6.1.6 Discussion

Capo was evaluating using a replay of my 2010 trace at UBC, focusing on the peak periods identified in Table 6.1 [SMW+10], and was found to reduce peak loads to 52.5% of their total, though much of this effect was later determined to be due to increasing request latency. However, Capo was still able to reduce total workload costs to 76% of those in the observed trace. When Capo was allowed to delay write-back to central storage, the gains were significantly larger, and the system was able to reduce peak and total I/O workloads to 38.1% and 28.6% of that of the system with no caching in place. A more thorough treatment of the Capo system and its evaluation can be found in the full paper [SMW+10].

My findings have two immediate implications. First, they provide an important characterization of VDI workloads, which are generally useful to the storage system and research community. Second, they show that tracing can reveal relatively simple approaches to improving system efficiency in widely used systems. Without tracing, the degree to which co-located virtual machines share content would be entirely hidden by the storage stack. Even with conventional tracing the degree to which content written by different VMs at the same offset is identical would be unclear. However, with detailed tracing and analysis I was able to largely determine potential for a cache before the system itself was built. I was also able to use my trace to ultimately evaluate Capo with trace replay. I revisit the further benefits of more detailed tracing with respect to Capo in Chapter 7.

6.2 Eliminating Wasted Effort

Next I consider a different application of tracing to provide introspection into virtual workloads – eliminating unwanted requests. As I have discussed, the storage stack of a virtualized system is deep. My tracing framework provides a novel lens through which a storage administrator can introspect into the workloads from a client perspective. In addition, the tracing tools I have developed are unusually aggressive in collecting information about each request.

Without these features a typical storage administrator has no mechanism to determine which client-side applications are contributing to the load on their system. In contrast, by including originating application for most requests, I can rank applications by their file system IOPS in each peak workload listed in Table 6.1.

Table 6.3 shows each peak load period in Table 6.1, and for each period, shows the 3 highest contributors to IOPS load in order, excluding requests that are serviced by the cache. I also excluded system and scvhost because, while they are generally large contributors, they often aggregate requests on behalf of many different applications and offer few opportunities for insight or administrative remedy. In addition, these services are frequently background tasks that have less direct impact on user-perceived performance.

In every peak Firefox is a significant contributor, and in many cases writes appear to originate from the creation of small temporary internet files. Based on this insight, I discuss how these accesses could be eliminated or reduced in Section 7.2.

	Time Period	Top Applications by IOPS (excludes system and scvhost)
1	Mon. 8:30am-8:45am	Search Indexer, Firefox, Sophos
2	Fri. 8:30am-9:00am	Firefox, Search Indexer, Sophos
3	Tue. 9:15am-9:30am	Defrag, Firefox, Search Indexer
4	Mon. 2:00pm-2:30pm	Firefox, Pidgin, Sophos
5	Tue. 2:40pm-3:00pm	Firefox, Defrag, Pidgin
6	Wed. 4:00pm-4:15pm	Firefox, Pidgin, Sophos

Table 6.3: Applications running in each of the peak load periods.

The second most commonly observed source of application I/O requests is Sophos, which is typically accessing storage to inspect files. It would be useful to determine if these I/O requests could be mitigated by performing virus scans centrally on gold master images, rather than accessing the overlay of every virtual desktop. Virus scanning is notorious in VDI environment for causing I/O storms, where otherwise idle machines all awaken and begin I/O intensive processes at the same time. Other frequently active storage consumers include Thunderbird, Pidgin, and Microsoft Outlook.

Two surprising entries in the contributor chart are defragmentation and the search indexer. Both are commonly listed on Virtualization "Optimization" guides as candidates to be disabled. The defragmentor was accidentally left enabled in these images, a fact I was able to communicate to the VDI team. Defragmentation of a virtual volume is not thought to provide any value, since the use of disk overlays and massive centralization of storage on a large disk array greatly complicates the linearity of access assumptions that generally lead one to defragment. The search indexer in this disk images should have been disabled by default, but could still have been invoked manually on the few machines where I saw it running. Automatic search

indexing in Windows is a useful feature, but can lead to I/O storms in a VDI cluster. In both cases these observations are obvious with tracing of the virtualized guest, but difficult to identify otherwise.

Multi-level tracing can also be deployed specifically to determine the cause of repeatable I/O bottlenecks. I have also used this tool to assist a colleague in identifying a bottleneck in the Windows boot process which was causing I/O storms 15 minutes after a Windows boot. The activity in this example was caused by SuperFetch [Cor14c], which must be disabled in the registry. This is straightforward to find when disk traces include the file and application associated with the request. Even though the I/O from each VM is issued as low priority, the collection of I/O activity from all the VMs proved problematic when attempting a 1000 VM boot up. Again, similar benefits of disk tracing with rich contextual information have been reported to me by others [Pra11] who have downloaded and deployed my tracing tool. These successes of tracing while aggressively gathering contextual information for each request all speak to the ability to helpfully eliminate problematic workloads, and the degree to which such workloads are hidden by virtualization and the storage stack. However, with tracing one can often eliminate such waste with relative ease. Conventional tracing tools inform users as to when periods of load are happening, but do not list the files or applications responsible.

6.3 On-Disk Fragmentation

My third example of the benefits of workload introspection draws from file system analysis of enterprise workstations, instead of virtual desktop workloads. In this section I consider the linearity of on-disk data layouts.

The behaviour and characteristics of magnetic disks continue to be a dominant concern in storage system optimization. It has been shown that file system performance changes over time, largely due to fragmentation [SS97], and that these effects can have dramatic impacts on system performance. While there is little doubt that these findings were true when they were published in 1997, I can find no more recent investigation into the effect and no study into the degree to which it practically impacts live systems today. As a consequence, it is difficult to know how realistic most published measurements of storage system performance are, because these measurements generally use synthetic datasets that, to a greater or lesser degree, attempt to simulate this effect.

My investigation into the file system structure of Windows desktops at Microsoft in 2009 can be used to investigate whether this concern in significant. Overall, I find fragmentation to be quite rare, occurring in only 4% of files. This lack of fragmentation in Windows desktops is due to the fact that a large fraction of files are not written after they are created and due to the defragmentor, which runs weekly by default. This is true for all of my scans other than the 17 that came from machines running Windows Server 2003.

I measure fragmentation in my dataset by recording the files' retrieval

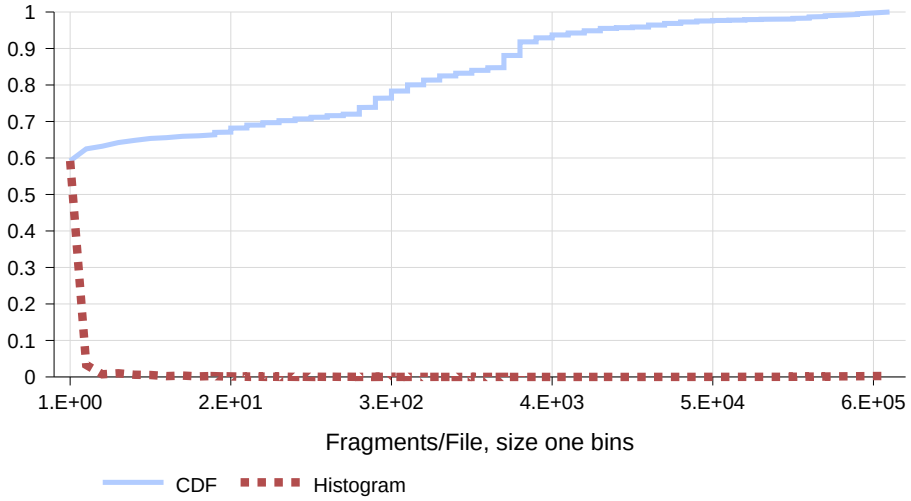

Figure 6.7: File Fragmentation. CDF and histogram of percentage of total fragments versus number of fragments per file.

pointers, which point to NTFS's data blocks. Retrieval pointers that are non-linear indicate a fragmented file. Figure 6.7 shows a CDF of the total number of fragments each measured as one of these non-linear breaks in a file's offset to Logical Block Address (LBA) mapping table.

The CDF shows that more than half of file fragments are in files that have exactly one fragment. Furthermore, even though fragmentation is not common, my results show that among files containing multiple fragments, fragments are very common. In fact, 25% of fragments are in files containing more than 170 fragments. The most highly fragmented files have a ".log" extension. From this I conjecture that these files are log files, which (if managed naively) may create a new fragment for each appending write. In this case, fragmentation is not likely to be impactful, because log files are not routinely read.

However, as I have shown in the prior chapter, file systems do contain many large disk image files which are internally complete file systems. This result says nothing about their internal structure. For these and other opaque file types fragmentation can occur at many different layers. If defragmentators are disabled, as is commonly recommended for virtual machines, virtual disks may have significant internal file system fragmentation. Furthermore, VHD formats themselves perform allocation from the files in linear 2 MB chunks. As disk formats become increasingly detached from the underlying media, the effect of fragmentation at these levels is poorly understod and warrants further study.

6.4 Summary

In a complex system, it is challenging to understand the behaviours of a workload and their effect on system performance. In this chapter I have investigated how efficiency lost as a result of this complexity can be better reclaimed though detailed tracing and analysis of system behaviour and have described how designers and administrators can use tracing to identify workload problems in specific systems or challenge untested assumptions in storage systems generally. In VDI systems, I have shown specifically how a simple cache can significantly increase the performance of a storage system, and how applications running hidden within a VM container can lead to wasted system effort. I have also shown that fragmentation at the file system layer of desktop workstations is not a significant concern today, even as there may be reason for increased concerns with fragmentation inside

opaque file types. In every case, workload introspection makes solutions (or in the case of fragmentation, the efficacy of existing solutions) obvious, where previously they were hidden.

More recently, Le, Huang, and Wang have noted that the nesting of file systems common in VDI and other virtualized environments can lead to surprising levels of write amplification and degraded performance [LHW12]. Like the observations in this chapter, this finding is simple and impactful, but hidden by the layers of a complex storage system. Tools for analysis and tracing that make wasted effort or optimization opportunities obvious provide a mechanism for administrators to routinely investigate their own workloads to ensure their efficiency. Such tools also present a resource for designers by making knowledge of the behaviours of real systems more widely available.

Chapter 7

Namespace Modelling

Enterprise workstation file systems are typically organized in a hierarchical structure of subdirectories and files contained within directories, up to the file system root. Within that hierarchy, some directories, for example a user's home directory, have special meaning or application within the context of the larger operating system. However, in most storage systems there is no special attention paid to the organization and structure of the namespace. From the storage system's perspective the contents of a user directory are usually no different from data elsewhere in the system. In contrast, I argue that more detailed analytic study and modelling of namespaces reveals new opportunities to improve storage system design.

In this chapter I argue this position from two perspectives. First, I characterize the organization of desktop workstation namespaces generally and discuss how they have evolved since the year 2000. I argue that this is helpful to the design of new storage features by using my results to test three assumptions made of namespaces in research prior to this work. I then consider a novel application of leveraging a model of file system structure from within a storage system operating over live data.

The sections in this chapter are as follows:

- In **Section 7.1** I present findings from my 2009 file system study to characterize the structure and contents of file system namespaces. I compare these results to assumptions made in prior research.

- In **Section 7.2** I describe a technique for using knowledge of file system namespace topology to decrease the overheads associated with performing writeback in a storage system cache. I also demonstrate one application of including file system namespace information in workloads, by using my 2010 study of VDI workloads at UBC to validate this system.

- In **Section 7.3** I summarize and discuss the contents of this chapter.

These examples both use tracing and analysis to improve our understanding of file system namespaces. In the former example, I apply knowledge of the namespaces to guide researchers and system designers. In the latter example I apply knowledge of the namespace to optimize the performance of a live system. These show how building models of the topology of file system namespaces is a particularly interesting subject for analysis, can be applied towards a better understanding of how our file systems are evolving, and can be leveraged as a tool to better optimize and tune our storage systems. I begin with a characterization of enterprise file system namespaces.

7.1 File System Namespace Characterization

The designers of file systems and file system features routinely make assumptions about the structure and organization of file system namespaces.

Sometimes these assumptions are implicit and other times not. For example, Huang et al have built a file system query mechanism that depends on sampling the namespace with a small number of depth first traversals. The stastical soundness of this technique depends on whether that small number of traversals can capture a representative sample of the namespace as a whole [HZW+11]. If the namespace is very heterogeneous it its hierarchy, then the sampling method must be made correspondingly more complicated, which runs counter to the project's objective of sampling the namespace with very low overhead.

Other examples include Murphy and Seltzer, who have questioned the merits of traditional hierarchical file system presentations entirely [SM09], based in part on the argument that file systems namespaces are growing more complex. In this work the authors argue that presenting a file system namespace as a hierarchy (irrespective of the underlying on-disk layout, which generally employs some hierarchy by necessity) is simply too confusing for end-users to manage. However, the exact measure and definition of complexity in this case is difficult to define accurately without quantitative measure.

Finally, Peek has addressed storage system treatment of file extensions, assuming a very long tail distribution of supported file types [Pee10]. His work argues that the growing problem of supporting infrequently used file types demands automated file system support for extracting file type-specific metadata. However, the underlying assumption that there is a growing long-tail of distributions was tested only on a dataset of limited size.

Quantifying these assumptions against a general dataset of significant

size, as I do in this section, provides valuable evidence in support of or contradictory to these claims and others, which will help designers build better, more useful features. I report on file organization and type in the following two subsections, relating each to the research assumptions above, and then summarizing both at the end of the section.

7.1.1 Namespace Complexity

I begin by characterizing the namespace topology in order to address the concerns of Murphy and Seltzer, that namespaces are becoming more complex. There is no single widely accepted definition of namespace complexity. It could be seen, for example, as a larger number of files, or a less homogeneous topology, or some combination of these and other factors. In this section I enumerate a series of observations drawn from my dataset to help define and serve as evidence for or against namespace complexity.

First, I consider the organization of files within directories, starting with the number of files and directories. A CDF of total directory counts per file system is shown in Figure 7.1, and shows both an increase in the total number of directories and a widening of the distribution of directory counts. The observed mean was approximately 36,000 directories per file system.

Similarly, a CDF of total file counts per file system is shown in Figure 7.2, which shows a very similar but slightly less dramatic trend. The mean file count was approximately 225,000 files, but a non-trivial fraction roughly (1:20) of systems I observed contained over a million files and the most populated file system served 3.4 million files.

In both cases, it is clear that in addition to the increase in file sizes shown

in Chapter 5, there are marked increases in the size of desktop workstation namespaces. This increase is largely unsurprising, but the measurement of the increase is nonetheless valuable, and places the remaining results of this section into better context.

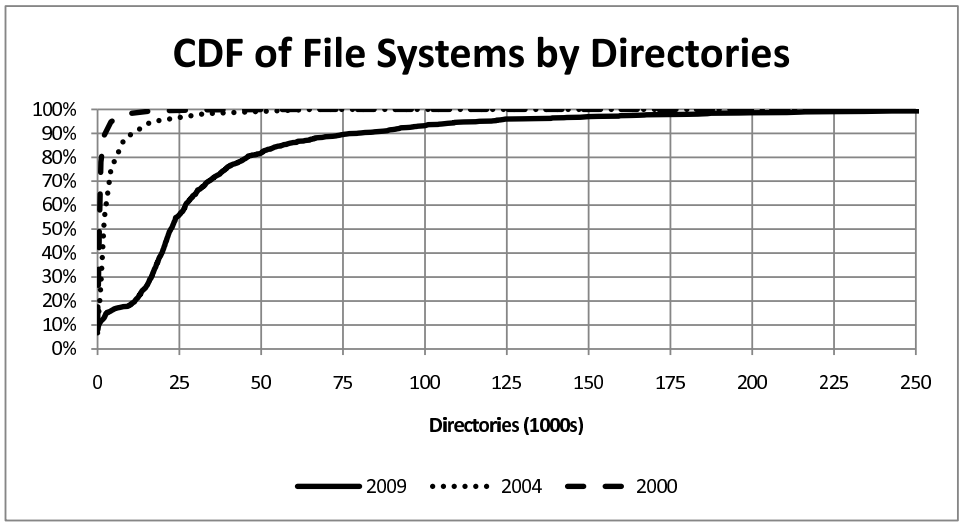

Figure 7.1: CDF of file systems versus directories.

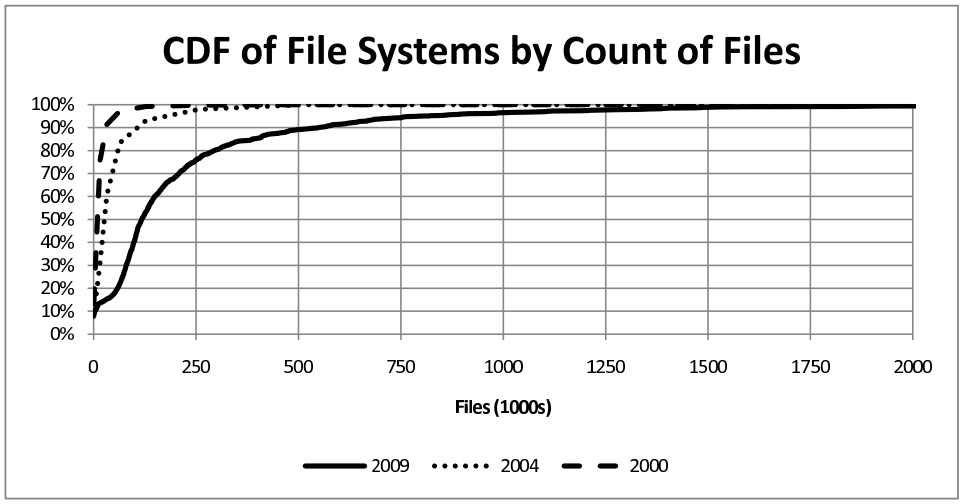

Figure 7.2: CDF of file systems versus files.

Next I consider the organization of files and directories within directories. Figure 7.3 shows the number of files per directory. While the change is small, it is clear that even as users in 2009 have many more files, they have fewer files per directory, with a mean of 6.25 files per directory.

Directories can also hold subdirectories. Figure 7.4 shows the distribution of subdirectories per directory. Since each directory is both a directory and subdirectory, the mean subdirectories per directory is necessarily one.[9] This means the fact that the distribution is slightly more skewed toward smaller sizes indicates that the directory structure is deeper with a smaller branching factor.

To add some context to this result, Figure 7.5 shows the histogram and CDF of files by directory depth for my on-disk data study. Similar results have not been published in any prior work to my knowledge, so this result cannot be compared to prior years. The histogram in this case is somewhat more informative than the CDF, as it shows a roughly normal distribution around a depth of 7 directories, with a spike at 3. The tail of files in deep directories is heavy, with 10% of files above depth 10 and some as deep as twice that. These findings further indicate that namespaces contain non-trivial amounts of data in a relatively small number of very deep paths.

[9]Ignoring that the root directory is not a member of any directory

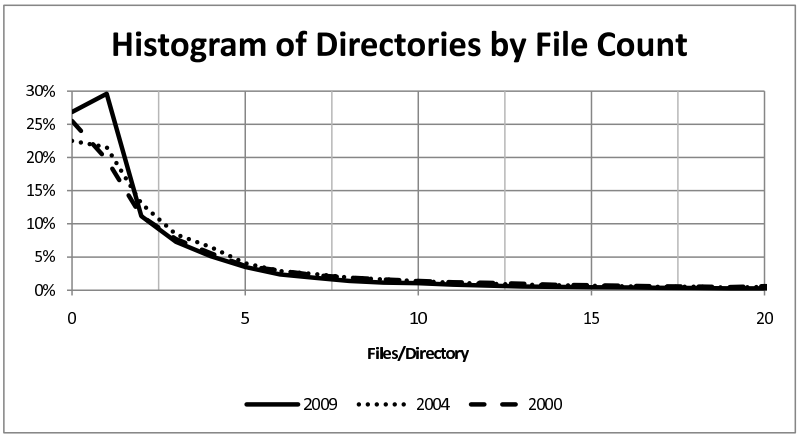

Figure 7.3: Histogram of number of files per directory.

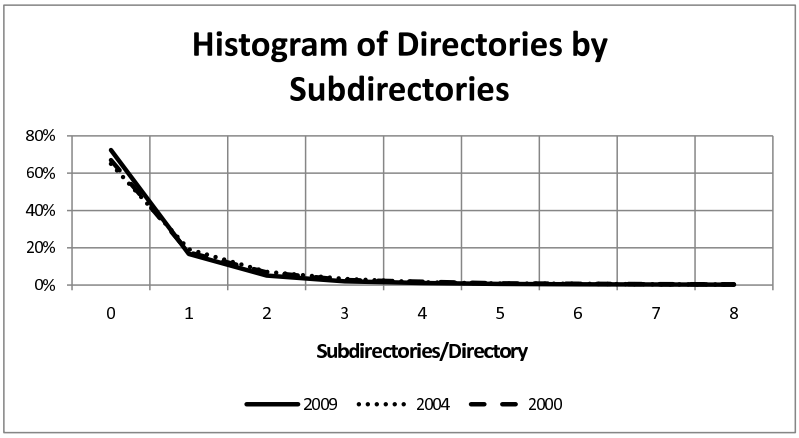

Figure 7.4: Histogram of number of subdirectories per directory.

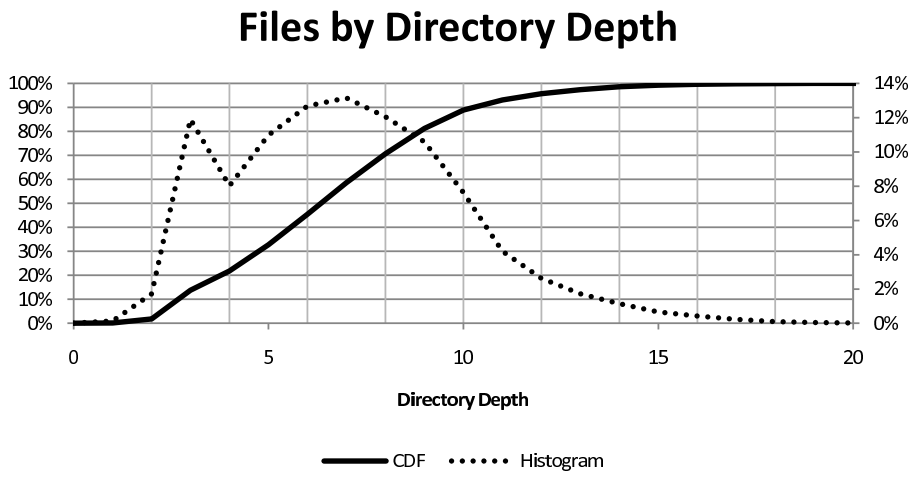

Figure 7.5: CDF of number of files versus depth in file system hierarchy.

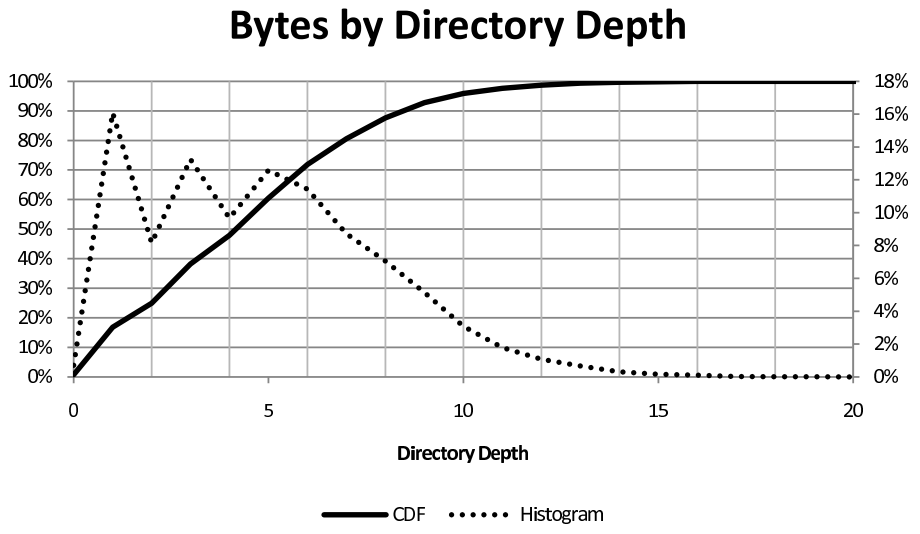

Figure 7.6: CDF of bytes versus depth in file system hierarchy.

Finally, I consider the how storage consumption is distributed among the storage hierarchy. Figure 7.6 shows the histogram and CDF of bytes by directory depth. This graph, combined with the previous result, shows that by capacity, larger files are higher in the namespace, with the many small files appearing at all depths of the namespace.

Overall, my dataset suggests that the namespace is changing in 4 ways:

1. There are significantly more files and directories overall.
2. The namespace branching factor is growing smaller.
3. The file system hierarchy is becoming deeper.
4. There are somewhat fewer files per directory.

These increases in namespace size and depth do lend some evidence to Murphy and Seltzer's argument that namespaces are growing more complex, which could argue for a change to file system organization, particularly if one assumes that the more rapid growth in directory counts is an attempt to manage the increasing file system count.

It is tempting to conclude that the exponential increase in file system count is causing an even larger increase in directory counts in an attempt to bring the burgeoning complexity back under control. This is a particularly attractive conclusion, because the effect appears to be uniformly distributed across the various depths of the file system namespace, which suggests that both users and software that autonomously populates the hierarchy are responding similarly. At the same time, fewer files per directory is most obviously a sign of decreased (or unchanging) complexity. I conclude that

while much of the weight of evidence does suggest an increase in namespace complexity, my results are mixed overall.

In addition, my results suggest that real namespaces contain data in a relatively small number of very deep paths, which contradicts the assumption of Huang et al of a nearly complete tree structure that can be sampled with a tractable number of depth-first traversals [HZW+11]. Figure 7.6 shows, similarly, that files beyond the average depth contain roughly 20% of the data in a typical file system.

7.1.2 File Types

I now turn my attention to the general popularity of file extensions, in an attempt to characterize this aspect of real namespaces and to assess Peek's assumption of a very long tail distribution of supported file types [Pee10].

Figure 7.7 shows a CDF of the number of files in the 2009 on-disk data study versus the number of unique extensions. Since 2000, there is a steady trend towards more diversity in file extensions. Still, the most common file extensions are used by most files in the system. In 2009 half of files had one of 14 extensions, whereas in 2000 half of files had one of 10 extensions.

Figure 7.8 details the ten most common extensions and their contribution towards the total number of files in the study. It is primary a testament to the large number of developer workstations in my dataset, but within that context reveals some interesting trends. GIF files (a web image format) and HTM files (the extension used for many web pages), particularly those created on Windows PCs, have both dropped dramatically in popularity both in the relative and absolute sense. At the same time, XML (the struc-

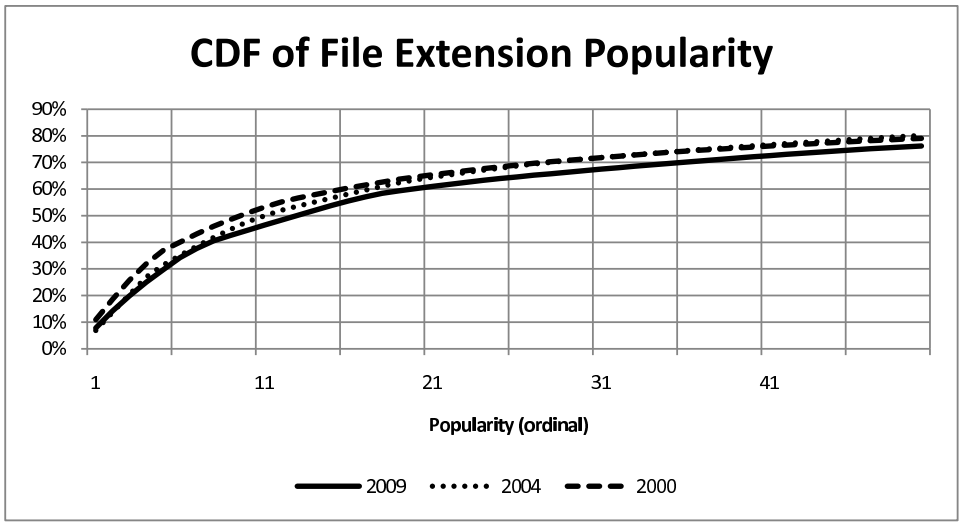

Figure 7.7: CDF of files versus extensions. Total file count versus number of file extensions for years 2009, 2004, and 2000 shown.

tured document layout format) and MANIFEST files, which is a specific XML-based format, have risen to meet dll (dynamic loaded library) and h files (C/C++ programming header) in popularity. Meanwhile, although cpp (C++ source files) remain as popular as ever, C programming source files have dropped off the list and been replaced with cs files (C# source files).

These findings, particularly the uniform increase in the rarity of file extensions shown in Figure 7.7, show that there is indeed a heavy tail of file extensions. It also shows that the trend is becoming more pronounced over time. These findings together do seem to support Peek's motivation [Pee10] for addressing support for rare file types.

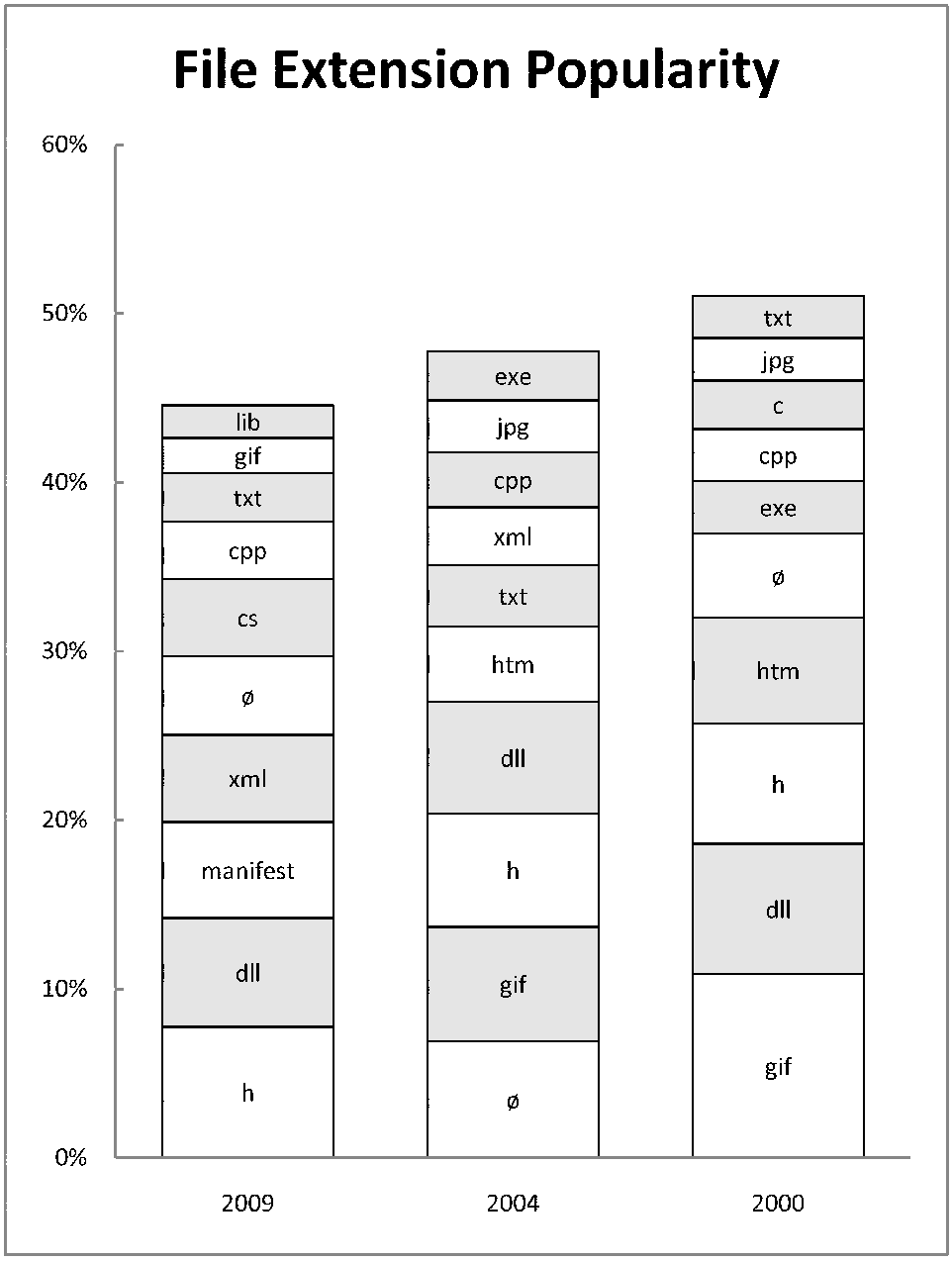

Figure 7.8: Percentage of files versus extension. Total file counts and file extensions for years 2009, 2004, and 2000 shown.

7.1.3 Discussion

My analysis of desktop workstation namespaces is a key resource for researchers who must frequently make assumptions based on characterizations of file system namespace structure. In a specific example, the long tail of directory depth relative to the average depth of files suggests that approaches to querying that use file system sampling [HZW+11] are likely to need very large sample sizes or must otherwise find ways to sample the deepest paths through the namespace. This result is also seen in other heavily skewed distributions, for example that of file size. I can confirm the assumptions of Peek [Pee10], that there is an increasingly long tail of file types. Finally, though the evidence is mixed, there is good support of Murphy and Seltzer's [SM09] view that namespaces are becoming more complex.

I have quantified the precise ways in which storage system namespaces are becoming more complex. My findings are that there are more files and directories, deeper namespaces, and more diverse file types. Only the number of files and subdirectories per directory suggest any simplification in structure (or at least a lack of increase in complexity). This information is important for enterprise storage designers and administrators, who must both contend with the management and performance challenges that come from deep and complex hierarchies and a wider array of file types. If this trend is to continue it seems likely that more support for understanding and searching file systems will be necessary. In the past, proposals such as Semantic File Systems [GJSO91], and query-based namespaces [GM99] have drawn research interest but little traction in general purpose file sys-

tems. It may be time to revisit some of these ideas in the context of modern workloads.

In the following section I consider the another application of modelling namespaces in desktop workstations – that of using the namespace to prioritize data placement.

7.2 Leveraging Namespaces with Differential Durability

Live storage systems typically pay little or no regard to the topology of namespaces. This section details my second application of modelling workstation namespaces, which is to use knowledge of the namespace to increase performance by prioritizing different data reliability policies. I call this approach *Differential Durability*, and have used it to reduce I/O load on a central storage system of up to 28.4% under simulation.

Differential Durability was inspired by an analysis of my 2010 study of VDI performance, which is presented below. I also describe how the feature was added to the Capo cache discussed in Chapter 6. Finally I use the results of that study as well as several new micro-benchmarks to evaluate the effectiveness of the feature.

Workload Characterization

Recall that during the trace collection of my 2010 study of VDI performance, I took the step to associate each disk-bound block-level request with the name and path of the accessed file. This multi-level analysis has proven

extremely valuable in understanding VDI-based workloads. In Figure 7.9 I show aggregate read and write operation count and throughput for the entire trace, which is very nearly exactly 2:1 write heavy in operation count and 2:1 read heavy in throughput. I also show where those requests are located inside the file system namespace. I chose the categories shown to highlight different uses of the file system, under the presumption that users view files in their personal directories different than they do, for example, the pagefile, or `Program Files` that are only written by an installer.

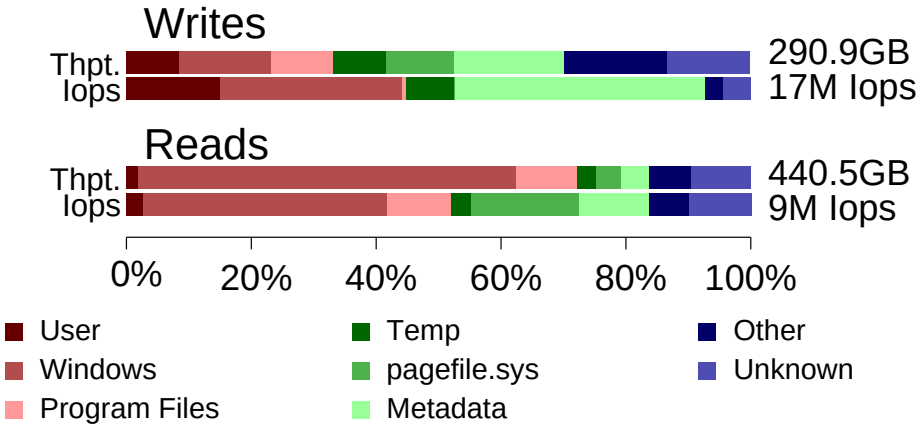

Figure 7.9: Size and amount of block-level writes by file system path.

The file-level information associated with the trace shows that metadata operations account for large portion of the requests. My current trace driver is unable to determine where within the namespace these operations occur, but it can determine the namespace location of most file-level operations. Files within directories typically managed by the operating system, such as `\Windows` and `\Program files` are also very frequently accessed. There are

fewer accesses to files in user directories and temporary files; most of the latter are to \Temporary Internet Files, as opposed to \Windows\Temp. These findings strongly contrast those of Vogels who's older study showed that 93% of *file-level* modification occurred in \User directories [Vog99]. From this I conclude that while a wide range of the namespace is accessed, it is not accessed uniformly, and access to data directly managed by users is rare. These findings are important because they suggest that applying data retention policies specific to the type of data being accessed may provide a potential benefit, as I will show.

Namespace Divergence

Similar to the per-VM total disk divergence measurement in Figure 6.4, one might wonder if the shared file system namespace diverges from the base image at different rates in different regions of the namespace.

Figure 7.10 plots the cumulative divergence for each VM in the cluster under observation, and divides that total among various components of the namespace. One can observe that the pagefile diverges immediately, then remains a constant size over time, as does the system metadata. Both these files are bounded in size. Meanwhile writes to \Windows and areas of the disk I cannot associate with any file continue to grow over the full week of the study. This finding supports my previous presumption that different locations within the namespace are accessed differently. I conclude that while writes occur everywhere in the namespace, they exhibit meaningful trends when categorized according to the destination.

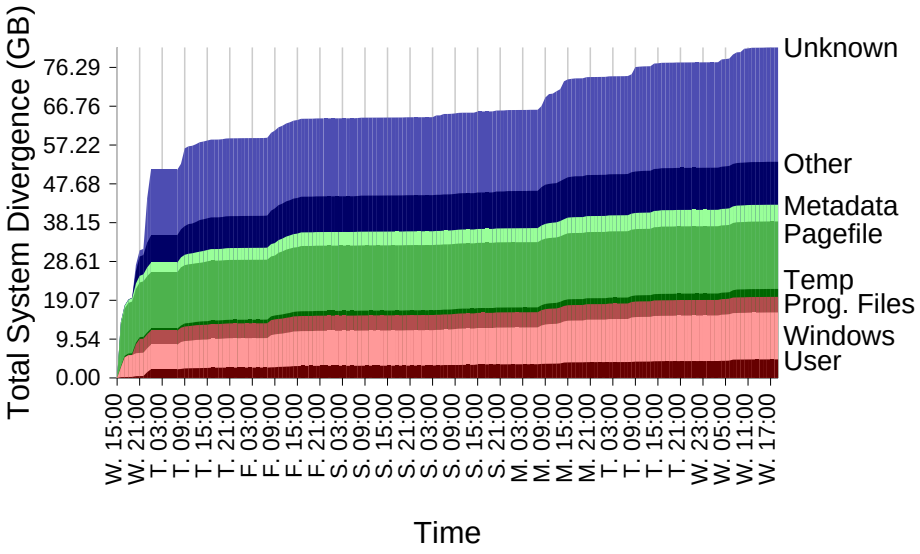

Figure 7.10: Total divergence versus time for each namespace category.

7.2.1 Differential Durability

Drawing from my findings on the non-uniformity of namespace access above, I propose a new model for delaying file system writeback as a performance optimization. Under a *Differential Durability* writeback policy, file-level accesses are assigned a policy that operates over the location of the modification within the namespace. A cache, like the Capo system described in Section 6.1.5, then uses these policies to determine how long to delay writeback, in order to lower both the peak and average load on central storage.

Recall that all major VDI providers have adopted the software update strategy proposed in The Collective [CZSL05], where user directories are isolated from the rest of the file system. Meanwhile modifications to the system image are performed on all VMs in one step by completely replacing the system images in the entire pool, leaving the user's data unmodified.

This impacts durability—any writes to the system portion (e.g., by updating the registry or installing software) will be lost. Differential Durability extends this notion to optimize for a write-heavy workload by allowing different write-back policies for different locations in the file system namespace.

Reducing Write-Back Costs

As a thought experiment, consider a user who installs Microsoft Word and uses it to create a short document. The install process creates temporary .cab files, which are deleted after installation. The installation itself writes to the Windows Registry and `Program Files` folder. Finally, the user's document will be saved to their desktop.

One might ask - *In the event of an unusual failure and crash, what level of loss is acceptable in exchange for efficiency?* To which one would almost certainly answer - *It depends.* The temporary files by their very nature are ephemeral and could be lost with no penalty. Contrastingly, it is reasonable to assume that the user document represents original work and should never be lost. Between these two extremes, the installation itself represents *some* work on behalf of the user, but a perfunctory installation could be repeated if necessary. It might be acceptable if the loss of such effort was limited to, for example, an hour or even a day. I call these three classes of durability requirements: none, full, and partial.

Based on this example, consider that there are some system files that need not be durably stored at all. These include files that are discarded on system restarts or can easily be reconstructed if lost. Writes to the pagefile, for example, represent nearly a tenth of the total throughput to central-

ized storage in my workload, as shown in Figure 7.9. These writes consume valuable storage and network bandwidth, but since the pagefile is discarded on system restart, durably storing this data provides no benefit. The additional durability obtained by transmitting these writes over a congested network to store them on highly redundant centralized storage provides no value because this data fate-shares with the local host machine and its disk. Many temporary files are used in the same way, requiring persistent storage only as long as the VM is running.

Differential Durability policies designate this data to local disk only, assigning it a write-back cache policy with an infinitely long write-back period. In the event of a hardware crash on a physical host, the VM will be forced to reboot, and the data can be discarded.

Other writes such as the modifications to the `Program Files` directory can benefit from delaying write back, even for relatively short periods. These delayed writebacks can occur during periods of relative inactivity, while many re-writes can be absorbed entirely, as I showed in Figure 6.6. In the unlikely event of a local disk failure, a user's experience of losing a single day's modification to the `Program Files` folder looks no different from the weekly reset to the base image, and in practice it can eliminate a significant number of writes to central storage.

Differential Durability on User Files

Modifications made to files in the user directories must be durable; users depend on these changes. The Differential Durability policy therefore uses write-through caching on these accesses, propagating all changed blocks im-

mediately to the centralized storage servers. However, this is not as simple as coarsely partitioning the file system at the root directory.

While much of the data on the `User` volume is important to the user and must have maximum durability, Windows, in particular, places some files containing system data in the User volume. Examples include log files, the user portion of the Windows registry, and the local and roaming profiles containing per-application configuration settings. Table 7.1 shows some paths on User volumes in Windows that can reasonably be cached with a write-back policy and a relatively long write-back period. Obviously, consideration must be used in selecting these policies so as to preserve user effort.

Design

Initially, I approached the problem of mapping these policies to write requests as one of request *tagging*, in which a driver installed on each virtual desktop would provide hints to the local cache about each write. While this approach is flexible and powerful, maintaining the correct consistency between file and file system metadata (much of which appears as opaque writes to the Master File Table in NTFS) under different policies is challenging. Furthermore, the tagging requires many file name lookups, which increases memory overhead for every open file. Instead, I have developed a simpler and better performing approach using existing file system features.

The path-based policies I use in my experiments can be seen in Table 7.1; naturally, these may be customized by an administrator. This list is provided (and used) as an example of conservative, well reasoned policies. I

did not attempt to tailor these policies to this specific workload, in fact, as the evaluation will show, several opportunities for improving performance further still exist. I provide these policies to a disk optimization tool that I run when creating a virtual machine image. The optimization tool also takes a populated and configured base disk image. For each of the two less-durable policies, it takes the given path and moves the existing data to one of two newly-created NTFS file systems dedicated to that policy. It then replaces the path in the original file system with a reparse point (Window's analogue of a symbolic link) to the migrated data. This transforms the single file system into three file systems with the same original logical view. Each of the three file systems are placed on a volume with the appropriate policy provided by the local disk cache. This technique is similar to the view synthesis in Ventana [PGR06], though I am the first to apply the technique with a local cache to optimize performance.

Path	Policy
\Program Files\	write-back
\Windows\	write-back
\Users\ProgramData\VMware\VDM\logs	write-back
\Users\$USER$\ntuser.dat	write-back
\Users\$USER$\AppData\local	write-back
\Users\$USER$\AppData\roaming	write-back
\pagefile.sys	no-write-back
\ProgramData\Sophos	no-write-back
\Temp\	no-write-back
\Users\$USER$\AppData\Local\Microsoft\Windows\Temporary Internet Files	no-write-back
Everything else, including user data and file system metadata	write-through

Table 7.1: Sample cache-coherency policies applied as part of Differential Durability optimization.

I appreciate that applying different consistency policies to files in a single logical file system may be controversial. The risk in doing so is that a crash or hardware failure results in a dependency between a file that is preserved and a file that is lost. Such a state could lead to instability; however, I are aware of no dependencies crossing from files with high durability requirements to those with lower durability requirements in practice. Further, I observe that this threat already exists in the production environment I studied, which overwrites system images with a common shared image on a weekly basis.

Evaluation

This section describes several micro-benchmarks that evaluate the effectiveness of Differential Durability in isolation of other features and provide a clearer mapping of end-user activity to observed writes. I applied the policies in Table 7.1 to several realistic desktop workloads. For each, I measured the portion of write requests that would fall under each policy category. I use synthetic user generated workloads here to highlight both the scale of, and the reason for, Differential Durability's improvement in the overall workload. A breakdown of writes by their associated policy for each workload is shown in Figure 7.11. Differential Durability was also incorporated into Capo and evaluated against the full workload, which I discuss subsequently.

Web Workload

My web workload is intended to capture a short burst of web activity. The user is made to open Facebook [10] with Microsoft Internet Explorer, log in,

[10] `http:\\www.facebook.com`

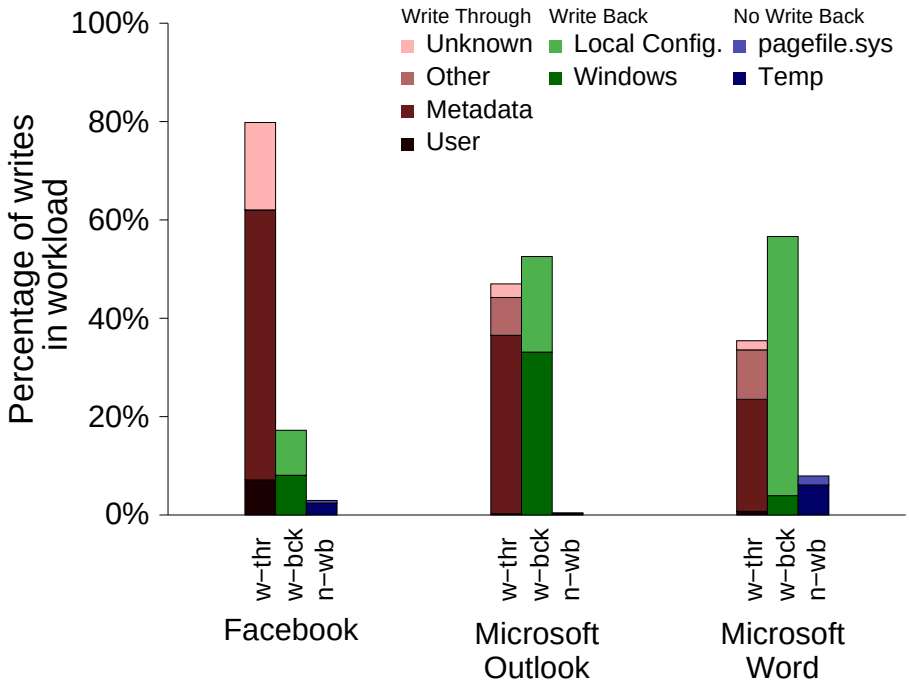

Figure 7.11: Percentage of writes by cache-coherence policy. Writes in each of the three micro-benchmarks are organized by governing cache-coherence policy.

and post a brief message to their account. They then log off and close the browser. The entire task lasts less than a minute. The workload consists of 8MB (43.6% by count) of writes and 25.3MB (56.4% by count) of reads. Recall that in Section 6.2 I showed that web browsing activity such as this is a significant contributor to the peak load periods I have been exploring.

In this short workload only a small but non-trivial improvement can be made. Local configuration changes such as registry, temp file, and cache updates are buffered for a short time, removing or delaying just over 20% of the operations.

Email Workload

My email workload is based on Microsoft Outlook. The user sends emails to a server I have configured to automatically reply to every message by sending back an identical message. Ten emails are sent and received in succession before the test ends. The workload consists of 63MB (39% by count) of writes and 148MB (61% by count) of reads matching well the characteristics observed in the trace.

Here the improvement is much more substantial. Although very few writes can be stored to local disk indefinitely, over half can be delayed in writing to centralized storage. This is due to Outlook's caching behaviour, which makes heavy use of the System and Application Data folders. User visible email files are all stored in .pst files included in the user category and safely stored, which I was able to verify by forcing a shutdown before delayed write-back. It is worth noting that many files in the Windows and Application Data categories are obvious temp files, but did not match any of the policies in Table 7.1. With more careful tuning, the policies could be further optimized for this workload.

Application Workload

My application workload is intended to simulate a simple editing task. The user opens Microsoft Word and creates a new document. The user also opens Wikipedia [11] in Microsoft Internet Explorer. She then proceeds to navigate to 10 random Wikipedia pages in turn, and copy the first paragraph of each

[11] `http:\\www.wikipedia.org`

into the word document, saving the document each time. Finally, the user closes both programs. The workload consists of 120MB (20.0% by count) of writes and 406MB (80.0% by count) of reads. In addition to simulating a typical simple multi-tasking activity, this test is intended to dirty memory.

Viewing many small pages creates a large number of small writes to temporary files and memory pressure[12] increases the pagefile usage. Both programs write significantly to System folders, leaving less than 36% of the workload to be issued as write-through.

Trace-Driven Workload

I also modified the disk image used in the trace replay of the Capo system in Section 6.1.5. With Differential Durability, Capo was able to reduce both peak and total loads by half, to 50.1% and 47.6% of their original totals. This is a respective improvement of 2.4% and 28.4% over a version of Capo without Differential Durability. A more detailed discussion is provided in the FAST 2011 Capo paper [SMW+10].

7.3 Summary and Discussion

The structure and organization of file system namespaces contain valuable information that is too often unavailable. From the perspective of designers and implementers who seek to understand file system structure there are very few public repositories from which to model namespaces. Outside of user-interaction studies, which are far more limited in scope, the charac-

[12]The guest was running Windows Vista with 1GB of RAM, 25% higher than the XenDesktop recommended minimum.

terization in Section 7.1 of this chapter form the only such resource made available in the past decade.

Applying models of desktop workstation namespaces to unchecked assumption in prior research yields a range of results. In some cases, new features do appear to be designed on assumptions which my datasets support well, but others do not. Sourcing relevant results in the study of real namespaces would help guide researchers and designers towards solving real problems in realistic ways.

As an example, I have showed that it is possible to leverage knowledge of file system structure in a live system to make trade offs between performance and durability. In this latter case, I was also able to evaluate the system's effectiveness on a live dataset because my 2010 trace of file system workload included multi-layer metadata, allowing me to correlate block-level access below the cache with file names and paths. Differential Durability is relatively easy to implement and simple to deploy once the insight to separate durability based on namespace is made clear by trace analysis. Further, it is quite effective at reducing complexity of storage systems as measured by requests made to central storage. An evaluation based on my traced workload showed a decrease in total request load of 28.4% due to Differential Durability alone.

Chapter 8

Conclusion

In this thesis I have detailed several advances in the measurement of data storage systems. In Chapter 4 I described the architecture of my data collection and analysis framework, and described two case studies in which it was applied to gather information about large scale systems in live deployment. Based on these case studies I have presented findings in three areas of storage system management and development.

In Chapter 5 I described how tracing and analysis can be applied to manage storage system capacities more effectively and characterized the duplication inherent in a general purpose dataset for the first time. Among my results, I find that while all forms of deduplication work, most of the advantage of performance intensive deduplication can be had by simpler approaches. I also find that selectively targeting the specific files where complex approaches work more effectively is likely to close the gap even further.

In Chapter 6 I explored three applications of workload introspection, which I define as the ability to gain insight into workloads via tracing and analysis. I showed how tracing was instrumental in the design, motivation, and evaluation of the Capo cache. I also list a number of cases in which more

detailed tracing than is typical has aided in their identification of problematic workloads. Finally I describe how I have used file system analysis to present the first empirical measurement of commercial defragmentation.

In Chapter 7 I described how modelling of file system namespaces can be used to better characterize our storage systems and the impacts on existing research, and also how it can be leveraged to enable new alternatives in balancing performance and durability. With Differential Durability writes are buffered in a cache for a controlled period of time in order to mitigate system load.

In addition to my analysis and datasets, the tools I have created are intended to be used for an organization to perform regular, persistent analysis of their workloads and systems, in order to facilitate simple and straightforward approaches to data management and mitigating the effects of growing system size and feature complexity. The suite of tools also presents a useful starting point for other researchers wishing to perform their own studies of system behaviour, in order to corroborate my findings, or more usefully, extend the relatively small body of existing public datasets on storage system workloads, metadata, and content.

Bibliography

[AAADAD11] Nitin Agrawal, Leo Arulraj, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Emulating goliath storage systems with David. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies*, FAST'11, Berkeley, CA, USA, February 2011. USENIX Association.

[AADAD09] Nitin Agrawal, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Generating realistic impressions for file-system benchmarking. In *Proceedings of the 7th USENIX Conference on File and Storage Technologies*, FAST'09, pages 125–138, Berkeley, CA, USA, February 2009. USENIX Association.

[ABDL07] Nitin Agrawal, William J. Bolosky, John R. Douceur, and Jacob R. Lorch. A five-year study of file-system metadata. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies*, FAST'07, Berkeley, CA, USA, February 2007. USENIX Association.

[ALR+12] Cristina L. Abad, Huong Luu, Nathan Roberts, Kihwal Lee, Yi Lu, and Roy H. Campbell. Metadata traces and work-

load models for evaluating big storage systems. In *Proceedings of the 2012 IEEE/ACM Fifth International Conference on Utility and Cloud Computing*, UCC '12, pages 125–132, Washington, DC, USA, 2012. IEEE Computer Society.

[AMW+03] Marcos K. Aguilera, Jeffrey C. Mogul, Janet L. Wiener, Patrick Reynolds, and Athicha Muthitacharoen. Performance debugging for distributed systems of black boxes. In *Proceedings of the 19st ACM Symposium on Operating Systems Principles*, SOSP'03, pages 74–89, New York, NY, USA, 2003. ACM.

[And09] Eric Anderson. Capture, conversion, and analysis of an intense NFS workload. In *Proccedings of the 7th USENIX Conference on File and Storage Technologies*, FAST '09, pages 139–152, Berkeley, CA, USA, February 2009. USENIX Association.

[Ass13] Storage Networking Industry Association. Traces, October 2013. `http://iotta.snia.org/traces/`.

[Ass14a] Storage Networking Industry Association. IOTTA repository, 2014. `http://iotta.snia.org/traces`.

[Ass14b] Storage Networking Industry Association. SSSI workload I/O capture program, 2014. `http://snia.org/forums/sssi/wiocp`.

[AWZ04] A. Aranya, C. P. Wright, and E. Zadok. Tracefs: A file system to trace them all. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, FAST'04, pages 129–143, Berkeley, CA, USA, March 2004. USENIX Association.

[BBU+09] Shivnath Babu, Nedyalko Borisov, Sandeep Uttamchandani, Ramani Routray, and Aameek Singh. DIADS: Addressing the "my-problem-or-yours" syndrome with integrated SAN and database diagnosis. In *Proceedings of the 7th USENIX Conference on File and Storage Technologies*, FAST '09, pages 57–70, Berkeley, CA, USA, February 2009. USENIX Association.

[BCG+07] Paramvir Bahl, Ranveer Chandra, Albert Greenberg, Srikanth Kandula, David A. Maltz, and Ming Zhang. Towards highly reliable enterprise network services via inference of multi-level dependencies. In *Proceedings of the 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '07, pages 13–24, New York, NY, USA, 2007. ACM.

[BCGD00] William J. Bolosky, Scott Corbin, David Goebel, and John R. Douceur. Single Instance Storage in Windows 2000. In *Proceedings of the 4th conference on USENIX Windows Systems Symposium*, pages 2–2, Berkeley, CA, USA, 2000. USENIX Association.

[BDIM04] Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. Using Magpie for request extraction and workload modelling. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, OSDI'04, pages 18–18, Berkeley, CA, USA, 2004. USENIX Association.

[BELL09] Deepavali Bhagwat, Kave Eshghi, Darrell D. E. Long, and Mark Lillibridge. Extreme binning: Scalable, parallel deduplication for chunk-based file backup. In *Proceedings of the 20th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, MASCOTS '99, pages 1–9, Washington, DC, USA, 2009. IEEE Computer Society.

[BGU+09] Medha Bhadkamkar, Jorge Guerra, Luis Useche, Sam Burnett, Jason Liptak, Raju Rangaswami, and Vagelis Hristidis. BORG: Block-ReORGanization for self-optimizing storage systems. In *Proceedings of the 7th USENIX Conference on File and Storage Technologies*, FAST'09, pages 183–196, Berkeley, CA, USA, February 2009. USENIX Association.

[BHK+91] Mary G. Baker, John H. Hartmart, Michael D. Kupfer, Ken W. Shirriff, and John K. Ousterhout. Measurements of a distributed file system, 1991.

[BKL+10] Doug Beaver, Sanjeev Kumar, Harry C. Li, Jason Sobel, and Peter Vajgel. Finding a needle in Haystack: Facebook's

photo storage. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation*, OSDI'10, pages 1–8, Berkeley, CA, USA, 2010. USENIX Association.

[Blo70] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13:422–426, 1970.

[CAK+04] Mike Y. Chen, Anthony Accardi, Emre K.c.man, Jim Lloyd, Dave Patterson, Armando Fox, and Eric Brewer. Path-based failure and evolution management. In *Proceedings of the 1st USENIX Symposium on Networked Systems Design and Implementation*, NSDI'04, pages 309–322, Berkeley, CA, USA, March 2004. USENIX Association.

[CAVL09] A.T. Clements, I. Ahmad, M. Vilayannur, and J. Li. Decentralized deduplication in SAN cluster file systems. In *Proceedings of the USENIX Annual Technical Conference*, ATEC'09, page 8, Berkeley, CA, USA, June 2009. USENIX Association.

[CCZ07] Anupam Chanda, Alan L. Cox, and Willy Zwaenepoel. Whodunit: Transactional profiling for multi-tier applications. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys'07, pages 17–30, New York, NY, USA, 2007. ACM.

[CGC11] Cornel Constantinescu, Joseph S. Glider, and David D.

Chambliss. Mixing deduplication and compression on active data sets. In *Data Compression Conference*, DCC'11, pages 393–402, 2011.

[Cor10a] Microsoft Corp. File systems, 2010.

[Cor10b] Microsoft Corp. Volume Shadow Copy Service, 2010.

[Cor13a] Microsoft Corp. Computer down not crash when..., October 2013. `http://support.microsoft.com/kb/2492505`.

[Cor13b] Microsoft Corporation. BackupRead function, November 2013. `http://msdn.microsoft.com/en-us/library/aa362509(VS.85).aspx`.

[Cor13c] Microsoft Corporation. Recommended backup schedule for system center 2012 - operations manager, November 2013. `http://technet.microsoft.com/en-us/library/hh278860.aspx`.

[Cor13d] Symantec Corporation. Recommended basic backup configurations when protecting Enterprise Vault 9.0 using the NetBackup 7.0 or later Enterprise Vault Agent., September 2013. `http://www.symantec.com/docs/TECH150237`.

[Cor14a] Microsoft Corporation. Legal and corporate affairs, 2014. `www.microsoft.com/en-us/legal/Default.aspx`.

[Cor14b] Microsoft Corporation. Load order groups and altitudes

for minifilter drivers, 2014. `http://msdn.microsoft.com/en-us/library/windows/hardware/ff549689.aspx`.

[Cor14c] Microsoft Corporation. Windows 7 & SSD: Defragmentation, SuperFetch, prefetch, 2014. `http://support.microsoft.com/kb/2727880`.

[CWM+14] Brendan Cully, Jake Wires, Dutch Meyer, Kevin Jamieson, Keir Fraser, Tim Deegan, Daniel Stodden, Geoffre Lefebvre, Daniel Ferstay, and Andrew Warfield. Strata: High-performance scalable storage on virtualized non-volatile memory. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies*, FAST'14, pages 17–31, Berkeley, CA, USA, February 2014. USENIX Association.

[CWO+11] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, Jaidev Haridas, Chakravarthy Uddaraju, Hemal Khatri, Andrew Edwards, Vaman Bedekar, Shane Mainali, Rafay Abbasi, Arpit Agarwal, Mian Fahim ul Haq, Muhammad Ikram ul Haq, Deepali Bhardwaj, Sowmya Dayanand, Anitha Adusumilli, Marvin McNett, Sriram Sankaran, Kavitha Manivannan, and Leonidas Rigas. Windows Azure Storage: A highly available cloud storage service with strong consistency. In *Proceedings of the 23th ACM Symposium on Operating Systems Princi-*

ples, SOSP'11, pages 143–157, New York, NY, USA, 2011. ACM.

[CZSL05] Ramesh Chandra, Nickolai Zeldovich, Constantine Sapuntzakis, and Monica S. Lam. The Collective: A cache-based system management architecture. In *Proceedings of the 2nd USENIX Symposium on Networked Systems Design and Implementation*, NSDI'05, pages 259–272, Berkeley, CA, USA, May 2005. USENIX Association.

[DB99] John R. Douceur and William J. Bolosky. A large-scale study of file-system contents. *Proceedings of the 1999 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, 27:59–70, May 1999.

[DDL+11] Wei Dong, Fred Douglis, Kai Li, Hugo Patterson, Sazzala Reddy, and Philip Shilane. Tradeoffs in scalable data routing for deduplication clusters. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies*, FAST'11, Berkeley, CA, USA, 2011. USENIX Association.

[DGH+09] Cezary Dubnicki, Leszek Gryz, Lukasz Heldt, Michal Kaczmarczyk, Wojciech Kilian, Przemyslaw Strzelczak, Jerzy Szczepkowski, Cristian Ungureanu, and Michal Welnicki. HYDRAstor: a scalable secondary storage. In *Proceedings of the 7th USENIX Conference on File and Storage Technolo-*

gies, FAST'09, pages 197–210, Berkeley, CA, USA, February 2009. USENIX Association.

[ELMS03] Daniel Ellard, Jonathan Ledlie, Pia Malkani, and Margo Seltzer. Passive NFS tracing of email and research workloads. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, FAST '03, pages 203–216, Berkeley, CA, USA, March 2003. USENIX Association.

[emIC+05] Michael Abd el malek, William V. Courtright Ii, Chuck Cranor, Gregory R. Ganger, James Hendricks, Andrew J. Klosterman, Michael Mesnier, Manish Prasad, On Salmon, Raja R. Sambasivan, Shafeeq Sinnamohideen, John D. Strunk, Eno Thereska, Matthew Wachs, and Jay J. Wylie. Ursa Minor: Versatile cluster-based storage. In *Proceedings of the 4th USENIX Conference on File and Storage Technologies*, FAST'05, pages 59–72, Berkeley, CA, USA, December 2005. USENIX Association.

[Fou13] The Apache Software Foundation. ZooKeeper does not recover from crash when disk was full, 2013. `https://issues.apache.org/jira/browse/ZOOKEEPER-1621`.

[GAM+07] Dennis Geels, Gautam Altekar, Petros Maniatis, Timothy Roscoe, and Ion Stoica. Friday: Global comprehension for distributed replay. In *Proceedings of the 4th USENIX Symposium on Networked Systems Design and Implementation*,

NSDI'07, Berkeley, CA, USA, April 2007. USENIX Association.

[GGtL03] Sanjay Ghemawat, Howard Gobioff, and Shun tak Leung. The Google file system. In *Proceedings of the 19st ACM Symposium on Operating Systems Principles*, SOSP'03, pages 29–43, New York, NY, USA, 2003. ACM.

[GJSO91] David K. Gifford, Pierre Jouvelot, Mark A. Sheldon, and James W. O'Toole, Jr. Semantic file systems. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, SOSP'91, pages 16–25, New York, NY, USA, 1991. ACM.

[GKA09] Ajay Gulati, Chethan Kumar, and Irfan Ahmad. Storage workload characterization and consolidation in virtualized environments. In *2nd International Workshop on Virtualization Performance: Analysis, Characterization, and Tools*, VPACT'09, 2009.

[GM99] Burra Gopal and Udi Manber. Integrating content-based access mechanisms with hierarchical file systems. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*, OSDI'99, pages 265–278, Berkeley, CA, USA, 1999. USENIX Association.

[Gur14] Scott Gurthrie. Windows Partner Conference (WPC). Keynote Presentation, July 2014.

[HBD+14] Tyler Harter, Dhruba Borthakur, Siying Dong, Amitanand Aiyer, Liyin Tang, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Analysis of HDFS under HBase: A Facebook messages case study. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies*, FAST'14, pages 199–212, Berkeley, CA, USA, February 2014. USENIX Association.

[HDV+12] Tyler Harter, Chris Dragga, Michael Vaughn, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. A file is not a file: Understanding the I/O behavior of Apple desktop applications. *ACM Transactions on Computer Systems (ToCS)*, 30(3):10:1–10:39, August 2012.

[HHS05] Hai Huang, Wanda Hung, and Kang G. Shin. FS2: Dynamic data replication in free disk space for improving disk performance and energy consumption. *SIGOPS Operating Systems Review*, 39(5):263–276, 2005.

[HMN+12] Danny Harnik, Oded Margalit, Dalit Naor, Dmitry Sotnikov, and Gil Vernik. Estimation of deduplication ratios in large data sets. In *Proceedings of the 28st IEEE Conference on Mass Storage Systems and Technologies*, MSST'12, pages 1–11, 2012.

[HS03] W. Hsu and A. J. Smith. Characteristics of I/O traffic in per-

sonal computer and server workloads. *IBM Systems Journal*, 42(2):347–372, April 2003.

[HSY05] Windsor W. Hsu, Alan Jay Smith, and Honesty C. Young. The automatic improvement of locality in storage systems. *ACM Transactions on Computer Systems (TOCS)*, 23(4), November 2005.

[HZW+11] H. Howie Huang, Nan Zhang, Wei Wang, Gautam Das, and Alexander S. Szalay. Just-in-time analytics on large file systems. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies*, FAST'11, pages 217–230, Berkeley, CA, USA, February 2011. USENIX Association.

[JHP+09] Weihang Jiang, Chongfeng Hu, Shankar Pasupathy, Arkady Kanevsky, Zhenmin Li, and Yuanyuan Zhou. Understanding customer problem troubleshooting from storage system logs. In *Proceedings of the 7th USENIX Conference on File and Storage Technologies*, FAST '09, pages 43–56, Berkeley, CA, USA, February 2009. USENIX Association.

[JM09] Keren Jin and Ethan L. Miller. The effectiveness of deduplication on virtual machine disk images. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, SYSTOR'09, pages 7:1–7:12, New York, NY, USA, 2009. ACM.

[JWZ05] N. Joukov, T. Wong, and E. Zadok. Accurate and effi-

cient replaying of file system traces. In *Proceedings of the 4th USENIX Conference on File and Storage Technologies*, FAST'05, pages 337–350, Berkeley, CA, USA, December 2005. USENIX Association.

[KDLT04] Purushottam Kulkarni, Fred Douglis, Jason LaVoie, and John M. Tracey. Redundancy elimination within large collections of files. In *Proceedings of the USENIX Annual Technical Conference*, ATEC'04, Berkeley, CA, USA, June 2004. USENIX Association.

[KJ08] Eric Koskinen and John Jannotti. BorderPatrol: Isolating events for black-box tracing. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, Eurosys'08, pages 191–203, New York, NY, USA, 2008. ACM.

[KLZ+07] Jonathan Koren, Andrew Leung, Yi Zhang, Carlos Maltzahn, Sasha Ames, and Ethan Miller. Searching and navigating petabyte-scale file systems based on facets. In *Proceedings of the 2nd International Workshop on Petascale Data Storage*, PDSW '07, pages 21–25, New York, NY, USA, 2007. ACM.

[KR10] Ricardo Koller and Raju Rangaswami. I/O deduplication: Utilizing content similarity to improve i/o performance. In *Proceedings of the 8th USENIX Conference on File and Stor-*

age Technologies, FAST'10, Berkeley, CA, USA, February 2010. USENIX Association.

[KTGN10] Michael P. Kasick, Jiaqi Tan, Rajeev Gandhi, and Priya Narasimhan. Black-box problem diagnosis in parallel file systems. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies*, FAST'10, pages 43–56, Berkeley, CA, USA, February 2010. USENIX Association.

[KU10] Erik Kruus and Cristian Ungureanu. Bimodal content defined chunking for backup streams. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies*, FAST'10, Berkeley, CA, USA, February 2010. USENIX Association.

[LCGC12] Maohua Lu, David D. Chambliss, Joseph S. Glider, and Cornel Constantinescu. Insights for data reduction in primary storage: a practical analysis. In *Proceedings of the 5th Annual International Systems and Storage Conference*, SYSTOR'12, page 17, New York, NY, USA, June 2012. ACM.

[LCSZ04] Zhenmin Li, Zhifeng Chen, Sudarshan M. Srinivasan, and Yuanyuan Zhou. C-Miner: Mining block correlations in storage systems. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, FAST'04, Berkeley, CA, USA, March 2004. USENIX Association.

[LEB+09] Mark Lillibridge, Kave Eshghi, Deepavali Bhagwat, Vinay

Deolalikar, Greg Trezise, and Peter Camble. Sparse indexing: Large scale, inline deduplication using sampling and locality. In *Proceedings of the 7th USENIX Conference on File and Storage Technologies*, FAST'09, pages 111–123, Berkeley, CA, USA, February 2009. USENIX Association.

[LEB13] Mark Lillibridge, Kave Eshghi, and Deepavali Bhagwat. Improving restore speed for backup systems that use inline chunk-based deduplication. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies*, FAST'13, Berkeley, CA, USA, February 2013. USENIX Association.

[LGW+08] Xuezheng Liu, Zhenyu Guo, Xi Wang, Feibo Chen, Xiaochen Lian, Jian Tang, Ming Wu, M. Frans Kaashoek, and Zheng Zhang. D3S: Debugging deployed distributed systems. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'08, pages 423–437, Berkeley, CA, USA, April 2008. USENIX Association.

[LHW12] Duy Le, Hai Huang, and Haining Wang. Understanding performance implications of nested file systems in a virtualized environment. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, FAST'12, page 8, Berkeley, CA, USA, February 2012. USENIX Association.

[LPGM08] Andrew W. Leung, Shankar Pasupathy, Garth Goodson,

and Ethan L. Miller. Measurement and analysis of large-scale network file system workloads. In *Proceedings of the USENIX Annual Technical Conference*, ATEC'08, pages 213–226, Berkeley, CA, USA, June 2008. USENIX Association.

[LSAH11] Quoc M. Le, Kumar SathyanarayanaRaju, Ahmed Amer, and JoAnne Holliday. Workload impact on shingled write disks: All-writes can be alright. In *Proceedings of the 2011 IEEE 19th Annual International Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems*, MASCOTS '11, pages 444–446, Washington, DC, USA, 2011. IEEE Computer Society.

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

[MB11] Dutch T. Meyer and William J. Bolosky. A study of practical deduplication. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies*, FAST'11, Berkeley, CA, USA, February 2011. USENIX Association.

[McK03] Marshall Kirk McKusick. Enhancements to the fast filesys-

tem to support multi-terabyte storage systems. In *Proceedings of the BSD Conference 2003*, BSDC'03, pages 9–9, Berkeley, CA, USA, 2003. USENIX Association.

[Mic09] Microsoft. Virtual hard disk image format specification, February 2009. `http://technet.microsoft.com/en-us/virtualserver/bb676673.aspx`.

[MJLF84] Marshall Kirk McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems (TOCS)*, 2:181–197, aug 1984.

[NDR08] Dushyanth Narayanan, Austin Donnelly, and Antony Rowstron. Write off-loading: Practical power management for enterprise storage. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, FAST'08, pages 17:1–17:15, Berkeley, CA, USA, February 2008. USENIX Association.

[NDT+08] D. Narayanan, A. Donnelly, E. Thereska, S. Elnikety, and A. Rowstron. Everest: Scaling down peak loads through i/o off-loading. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation*, OSDI'08, pages 15–28, Berkeley, CA, USA, 2008. USENIX Association.

[OCH+85] John Ousterhout, Herve Da Costa, David Harrison, John A. Kunze, Mike Kupfer, and James G. Thompson. A trace-

driven analysis of the UNIX 4.2 BSD file system. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles*, SOSP'85, pages 25–34, New York, NY, USA, 1985. ACM.

[Pee10] Daniel Peek. TrapperKeeper: The case for using virtualization to add type awareness to file systems. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Storage and File Systems*, HotStorage'10, Berkeley, CA, USA, 2010. USENIX Association.

[PGR06] Ben Pfaff, Tal Garfinkel, and Mendel Rosenblum. Virtualization aware file systems: Getting beyond the limitations of virtual disks. In *Proceedings of the 3rd USENIX Symposium on Networked Systems Design and Implementation*, NSDI'06, pages 353–366, Berkeley, CA, USA, May 2006. USENIX Association.

[PL10] N. Park and D.J. Lilja. Characterizing datasets for data deduplication in backup applications. In *Proceedings of the IEEE International Symposium on Workload Characterization*, IISWC'10, 2010.

[PP04] Calicrates Policroniades and Ian Pratt. Alternatives for detecting redundancy in storage systems data. In *Proceedings of the USENIX Annual Technical Conference*, ATEC'04, Berkeley, CA, USA, June 2004. USENIX Association.

[Pra11] Ian Pratt. Co-founder of Bromium. Personnal Communication, Nov 2011.

[QD02] Sean Quinlan and Sean Dorward. Venti: A new approach to archival data storage. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, FAST'02, page 7, Berkeley, CA, USA, January 2002. USENIX Association.

[Rab81] Michael Rabin. Fingerprinting by random polynomials. In *Tech Report TR-CSE-03-01*. Center for Research In Computing Technology, Harvard University, 1981.

[RBM13] Ohad Rodeh, Josef Bacik, and Chris Mason. BTRFS: The linux B-tree filesystem. *ACM Transactions on Storage (ToS)*, 9(3):9:1–9:32, August 2013.

[Riv92] R. Rivest. The MD5 message-digest algorithm, 1992. `http://tools.ietf.org/rfc/rgc1321.txt`.

[RKBH13] Kai Ren, YongChul Kwon, Magdalena Balazinska, and Bill Howe. Hadoop's adolescence: An analysis of Hadoop usage in scientific workloads. *Proceedings of the VLDB Endowment*, 6(10):853–864, August 2013.

[RLA00] Drew Roselli, Jacob R. Lorch, and Thomas E. Anderson. A comparison of file system workloads. In *Proceedings of the USENIX Annual Technical Conference*, ATEC'00, Berkeley, CA, USA, June 2000. USENIX Association.

[RSI12] Mark Russinovich, David Solomon, and Alex Ionescu. Developer Reference. Microsoft Press, 2012.

[SAF07] Ya-Yunn Su, Mona Attariyan, and Jason Flinn. AutoBash: Improving configuration management with operating system causality analysis. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles*, SOSP'07, pages 237–250, New York, NY, USA, 2007. ACM.

[SAGL06] Mirko Steinle, Karl Aberer, Sarunas Girdzijauskas, and Christian Lovis. Mapping moving landscapes by mining mountains of logs: novel techniques for dependency model generation. In *Proceedings of the 32nd international conference on very large data bases*, VLDB '06, pages 1093–1102. VLDB Endowment, 2006.

[Sat81] M. Satyanarayanan. A study of file sizes and functional lifetimes. In *Proceedings of the 8th ACM Symposium on Operating Systems Principles*, SOSP'81, pages 96–108, New York, NY, USA, 1981. ACM.

[Sch10] Erik Scholten. How to: Optimize guests for VMware View, July 2010. `http://www.vmguru.nl/wordpress/2010/07/how-to-optimize-guests-for-vmware-view`.

[SM09] Margo Seltzer and Nicholas Murphy. Hierarchical file systems are dead. In *Proceedings of the 12th USENIX Confer-*

ence on Hot Topics in Operating Systems, HotOS'09, Berkeley, CA, USA, 2009. USENIX Association.

[SMW+10] Mohammad Shamma, Dutch T. Meyer, Jake Wires, Maria Ivanova, Norman C. Hutchinson, and Andrew Warfield. Capo: Recapitulating storage for virtual desktops. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies*, FAST'10, Berkeley, CA, USA, February 2010. USENIX Association.

[SPGR08] Mark A. Smith, Jan Pieper, Daniel Gruhl, and Lucas Villa Real. IZO: Applications of large-window compression to virtual machine management. In *Proceedings of the 22nd Conference on Large Installation System Administration Conference*, LISA'08, pages 121–132, Berkeley, CA, USA, 2008. USENIX Association.

[SS97] Keith A. Smith and Margo I. Seltzer. File system aging. increasing the relevance of file system benchmarks. *Proceedings of the 1997 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, 25:203–213, June 1997.

[SZDR+11] Raja R. Sambasivan, Alice X. Zheng, Michael De Rosa, Elie Krevat, Spencer Whitman, Michael Stroucken, William Wang, Lianghong Xu, and Gregory R. Ganger. Diagnosing performance changes by comparing request flows. In *Pro-*

ceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation, NSDI'11, Berkeley, CA, USA, 2011. USENIX Association.

[TBZS11] Vasily Tarasov, Saumitra Bhanage, Erez Zadok, and Margo Seltzer. Benchmarking file system benchmarking: It *is* rocket science. In *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems*, HotOS'11, pages 9–9, Berkeley, CA, USA, 2011. USENIX Association.

[THKZ13] Vasily Tarasov, Dean Hildebrand, Geoff Kuenning, and Erez Zadok. Virtual machine workloads: The case for new benchmarks for NAS. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies*, FAST'13, Berkeley, CA, USA, February 2013. USENIX Association.

[TJWZ] A. Traeger, N. Joukov, C. P. Wright, and E. Zadok. A nine year study of file system and storage benchmarking. *ACM Transactions on Storage (TOS)*, 4(2):25–80.

[TKM+12] V. Tarasov, K. S. Kumar, J. Ma, D. Hildebrand, A. Povzner, G. Kuenning, and E. Zadok. Extracting flexible, replayable models from large block traces. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, FAST'12, Berkeley, CA, USA, February 2012. USENIX Association.

[TMB+12] V. Tarasov, A. Mudrankit, W. Buik, P. Shilane, G. Kuen-

ning, and E. Zadok. Generating realistic datasets for deduplication analysis. In *Proceedings of the USENIX Annual Technical Conference*, ATEC'12, Berkeley, CA, USA, June 2012. USENIX Association.

[UAA+10] Cristian Ungureanu, Benjamin Atkin, Akshat Aranya, Salil Gokhale, Stephen Rago, Grzegorz Cakowski, Cezary Dubnicki, and Aniruddha Bohra. HydraFS: A high-throughput file system for the HYDRAstor content-addressable storage system. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies*, FAST'10, Berkeley, CA, USA, February 2010. USENIX Association.

[VKUR10] Akshat Verma, Ricardo Koller, Luis Useche, and Raju Rangaswami. SRCMap: Energy proportional storage using dynamic consolidation. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies*, FAST'10, Berkeley, CA, USA, February 2010. USENIX Association.

[VMW10] VMWare. Virtual machine disk format (VMDK), July 2010. `http://www.vmware.com/technical-resources/interfaces/vmdk.html`.

[Vog99] Werner Vogels. File system usage in Windows NT 4.0. In *Proceedings of the 17st ACM Symposium on Operating Systems Principles*, SOSP'99, pages 93–109, New York, NY, USA, 1999. ACM.

[WA93] Randolph Y. Wang and Thomas E. Anderson. xFS: A wide area mass storage file system. Technical report, Berkeley, CA, USA, 1993.

[WDQ+12] Grant Wallace, Fred Douglis, Hangwei Qian, Philip Shilane, Stephen Smaldone, Mark Chamness, and Windsor Hsu. Characteristics of backup workloads in production systems. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, FAST'12, Berkeley, CA, USA, February 2012. USENIX Association.

[WHADAD13] Zev Weiss, Tyler Harter, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. ROOT: Replaying multithreaded traces with resource-oriented ordering. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, SOSP'13, pages 373–387, New York, NY, USA, 2013. ACM.

[WJK+05] C. P. Wright, N. Joukov, D. Kulkarni, Y. Miretskiy, and E. Zadok. Auto-pilot: A platform for system software benchmarking. In *Proceedings of the USENIX Annual Technical Conference*, ATEC'05, pages 175–187, Berkeley, CA, USA, April 2005. USENIX Association.

[WXH+04] Feng Wang, Qin Xin, Bo Hong, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Tyce T. Mclarty. File system workload analysis for large scale scientific computing applications. In *Proceedings of the 21st IEEE / 12th NASA God-*

dard Conference on Mass Storage Systems and Technologies, MSST'04, pages 139–152, 2004.

[YBG+10] Neeraja J. Yadwadkar, Chiranjib Bhattacharyya, K. Gopinath, Thirumale Niranjan, and Sai Susarla. Discovery of application workloads from network file traces. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies*, FAST'10, Berkeley, CA, USA, February 2010. USENIX Association.

[YMX+10] Ding Yuan, Haohui Mai, Weiwei Xiong, Lin Tan, Yuanyuan Zhou, and Shankar Pasupathy. SherLog: Error diagnosis by connecting clues from run-time logs. In *Proceedings of the 15th Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'10, pages 143–154, New York, NY, USA, 2010. ACM.

[YZP+11] Ding Yuan, Jing Zheng, Soyeon Park, Yuanyuan Zhou, and Stefan Savage. Improving software diagnosability via log enhancement. In *Proceedings of the 16th Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'11, pages 3–14, New York, NY, USA, 2011. ACM.

[ZLP08] Benjamin Zhu, Kai Li, and Hugo Patterson. Avoiding the disk bottleneck in the Data Domain deduplication file system. In *Proceedings of the 6th USENIX Conference on*

File and Storage Technologies, FAST'08, pages 18:1–18:14, Berkeley, CA, USA, February 2008. USENIX Association.

[ZS99] Min Zhou and Alan Jay Smith. Analysis of personal computer workloads. In *Proceedings of the 7th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, MASCOTS '99, pages 208–, Washington, DC, USA, 1999. IEEE Computer Society.

Appendix A

Selected Database SQL Queries

A.1 Mean File System Fullness

```
select 100 - (SUM(percentfree) / COUNT(*)) from scans
```

A.2 File Systems by Count of Files

```
select FileCount/1000, COUNT(*)
    from scans group by FileCount/1000
    order by FileCount/1000
```

A.3 Whole File Duplicates by File Extension

```
select top 50
      a.extension, SUM(a.size), COUNT(*), AVG(a.size),
      AVG(cast(b.CountOfDuplicates as bigint))
    from files as a inner join
```

```
  files_with_whole_file_duplicates as b
on a.FileSerial = b.FileSerial
group by extension
order by SUM(a.size) desc
```

Appendix B

Scanner Output Format

```
-=-Per-System Data-=-
[Standard Stream] or [Backup Stream]
Username (hashed)
Hostname (hashed)
System Directory Location
Current Time
Operating Ssytem
Processor Architecture
Processor Level
Processor Revision
Number of Processors
Total Physical Pages
Size of Page File
Total Virtual Memory
Volume name
Volume serial number
Volume max component length
```

```
File system flags from GetVolumeInformation()

File system format from GetVolumeInformation()

Sectors per cluster

Bytes per sector

Free clusters

Total clusters

Contents of NTFS_VOLUME_DATA_BUFFER

File system Size

File System Version


-=-Per-File Data-=-

Directory name:length (hashed)

File name:length (hashed)

extension:0 hashed (hashed)

namespace depth

file size

file attributes flag (in hex)

file id

number of hard links to file

the reparse flag status (in hex)

Creation time

last access time

last modification time

{      If the file is non-linear on disk

  SV:X => X= Location of start Vcn
```

```
  V:X:L:Y => X=Location of next Vcn, Y=Lcn
   ...
  V:X:L:Y => X=Location of next Vcn, Y=Lcn
}
{      If the file is sparse
  A:X:Y => X=sparse allocation offset, Y=length
   ...
  A:X:Y => X=sparse allocation offset, Y=length
}
Hash of data:chunk size
...
Hash of data:chunk size


-=-Log footer format-=-
LOGCOMPLETE
Completion time from time()
```