UNOBTUSIVE STORAGE MAINTENANCE

by

George Amvrosiadis

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto

© Copyright 2016 by George Amvrosiadis
Abstract

Unobtrusive Storage Maintenance

George Amvrosiadis
Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto
2016

Storage systems rely on maintenance tasks, such as backup and defragmentation, to provide a multitude of guarantees that range from data availability and security, to good performance. These tasks access large amounts of data and can significantly impact foreground applications. The work presented as part of this thesis outlines the challenges of storage maintenance and provides mechanisms that lower its impact, allowing it to be performed more efficiently.

Maintenance operations act as insurance for storage systems. To avoid the prohibitive cost incurred in their absence, storage operators pay the price of performing them periodically. In the case of reliability, for example, we find that the error characteristics of hard disk drives in Google’s data centers are diverse across models and operating environments. Due to this unpredictability, the use of periodic maintenance becomes unavoidable. Furthermore, recent research efforts on disaster prediction depend on metrics collected through periodic maintenance operations.

Unfortunately, periodic storage maintenance requires data to be accessed multiple times. As a case in point, we find that customers of Symantec and Veritas schedule backups multiple times a week. This often results in excess maintenance work that fails to complete within the scheduled downtime periods. At this point, foreground workload impact becomes unavoidable in order to maintain the same reliability guarantees.

To lessen this impact, maintenance I/O can be scheduled only when storage devices are otherwise idle. To achieve this, idleness must be predicted accurately. Fortunately, our extensive study of disk workloads reveals characteristics that allow idleness to be predicted accurately by simple models.

Given the number of maintenance tasks available today, idle time can prove insufficient. For such scenarios we present Duet, a framework that provides notifications to tasks about relevant changes in the system cache. Using this mechanism, tasks can reduce their I/O by prioritizing processing of cached data. Data can be cached either due to other maintenance tasks requesting it previously, or due to overlapping foreground I/O activity. In other words, tasks running concurrently can implicitly collaborate, collectively reducing the I/O required for maintenance and, by extension, the idle time needed to execute it.
To my parents
Acknowledgements

I jumped into the Ph.D. program directly after finishing my B.Sc. in Greece. As a result, graduate school has been a transformative experience on multiple levels: scholarly, culturally, and personally. Without a doubt, this transition would have been almost impossible without some incredible, self-less people: my parents Lia Kefala and Panos Amvrosiadis, my brother Thanos, Nada Fakhoury, Thanos Gousis, Makis Iosifidis, Thodoris Athanasiadis, Nikitas Efimidis, and Patrick Sin-Chan. Your support and understanding has been a steady boost through tough times, which were necessary to make this work so much more rewarding and fun.

I am grateful to Ashvin Goel, Angela Demke Brown, and Bianca Schroeder for taking up the challenging task of supervising me throughout different parts of my Ph.D., and for helping me develop a diverse skill set. You can rest assured that I carry your voices in my head, which is often a blessing and sometimes a curse.

I am also thankful to Medha Bhadkamkar for being an amazing collaborator and manager during my two-year tenure in Symantec and Veritas. She empirically demonstrated to me how one can balance excellent management and interpersonal skills in an industry setting, all while allowing creative freedom.

Every other person I have had the pleasure to collaborate with, or receive advice from, also deserves a big thanks: Alina Oprea, Ryan Johnson, Ioan Stefanovici, Nosayba El-Sayed, Andy Hwang, Francis Deslauriers, Peter McCormick, and all other members of the Systems and Networks lab at the University of Toronto. I have also been lucky to work with many brilliant undergraduate students, who patiently helped me improve my own management skills: Max Holden, Patrick J. Payne, Abdi Dahir, Suvanjan Mukherjee, Panay U. Jain, William Kingsford, Kasra Kyanzadeh, William Cui, and Hao (Dino) Liu.

That’s about it. Now let’s get to work.
Collaborative Work

A portion of the work presented as part of this thesis has been carried out in collaboration with people from the University of Toronto, Veritas Labs, Symantec Research Labs, and the RSA Laboratories of EMC. This statement lists the contributions of other researchers in detail.

The work presented in Section 2.1 was carried out in collaboration with Nosayba El-Sayed, Ioan Stefanovici, and Andy Hwang under the supervision of Prof. Bianca Schroeder. A full paper of our work was published at the ACM SIGMETRICS conference in 2012 [68].

The work presented in Sections 2.2 and 2.3 was conducted at Veritas Labs and Symantec Research Labs, respectively. In both cases, the work was carried out in collaboration with Dr. Medha Bhadkamkar. The results presented are a selection of the work published at the USENIX Annual Technical Conferences in 2015 [9] and 2016 [10].

The work presented in Chapter 3 was carried out in collaboration with Dr. Alina Oprea of EMC's RSA Laboratories, under the supervision of Prof. Bianca Schroeder. A full paper on this work was presented at the IEEE Dependable Systems and Networking conference in 2012 [12].

The work presented in Chapter 4 was completed under the supervision of Prof. Ashvin Goel and Prof. Angela Demke Brown, with the help of several students. The garbage collection case study for Duet’s evaluation was completed with the help of Max Holden, Pranay U. Jain, and Suvanjan Mukherjee. Part of the workload access pattern analysis (Figure 4.1) was carried out by Abdi Dahir. Finally, the extensions of the opportunistic work model described in Section 4.5 have been developed in collaboration with Francis Deslauriers, Peter McCormick, and Patrick J. Payne. Parts of this work were presented at the ACM Symposium on Operating Systems Principles in 2015 [11], and at the USENIX HotStorage workshop in 2016 [59].
## Contents

1 Introduction

1.1 Storage maintenance in the field .................................................. 1
1.2 Scheduling maintenance I/O ......................................................... 2
1.3 Opportunistic storage maintenance ............................................... 3
1.4 Thesis contributions .................................................................. 3
1.5 Thesis organization .................................................................. 4

2 Storage maintenance in the field ........................................... 5

2.1 The importance of periodic maintenance ................................. 5
2.1.1 Case study: Choosing between energy efficiency and reliability 6
2.1.2 Temperature and hard disk drives ......................................... 7
2.1.3 Implications on maintenance frequency ............................... 16
2.2 Finding time for maintenance ................................................... 17
2.2.1 Case study: Understanding how enterprise backups fail ......... 18
2.2.2 Veritas dataset description ....................................................... 19
2.2.3 Error characteristics .............................................................. 19
2.3 Too much work, too little time .................................................. 24
2.3.1 Case study: Characterizing enterprise backup system workloads 25
2.3.2 Anatomy of modern backup systems .................................... 26
2.3.3 Symantec dataset description ............................................... 27
2.3.4 Job scheduling ................................................................. 28
2.3.5 Backup data growth ............................................................. 32

3 Scheduling maintenance I/O ..................................................... 37

3.1 Understanding idleness .............................................................. 39
3.1.1 Insights from the statistical analysis of traces .................. 39
3.1.2 Profiting from idleness .......................................................... 44
3.1.3 Sizing up throughput opportunities .................................. 47
3.2 Implementing a scrubber ......................................................... 49
3.2.1 Scrub requests in SCSI/SAS vs. ATA/SATA .................... 49
3.2.2 The Completely Fair Queueing (CFQ) I/O scheduler ....... 50
3.2.3 User space, or kernel space? ............................................. 50
3.3 Getting to the bad sector at the right time .............................. 52
3.3.1 Staggered versus sequential scrubbing ............................ 52
6 Ongoing and future work

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1</td>
<td>Opportunistic computational frameworks</td>
<td>98</td>
</tr>
<tr>
<td>6.2</td>
<td>Enhancing file notification frameworks</td>
<td>99</td>
</tr>
<tr>
<td>6.3</td>
<td>Next-generation backup systems</td>
<td>99</td>
</tr>
</tbody>
</table>

7 Conclusion

Bibliography
List of Tables

2.1 Hard disk failure data used in our analysis, collected from Google data centers. .......... 8
2.2 Latent sector error data used in our analysis, collected from Google data centers. .......... 8
2.3 Specifications of the hard disk models used in our experiments. .............................. 14
2.4 Summary of our most important observations regarding backup system job errors. .......... 18
2.5 Breakdown of job types in our dataset. ................................................................. 19
2.6 Characteristics of the different types of backup systems in our dataset. The number of jobs and lifespan refer to averages across systems. ......................................................... 20
2.7 Summary of our most important observations regarding backup system workloads. ........ 26
2.8 Telemetry report metrics used in the study. ............................................................ 28
2.9 Breakdown of all jobs in the dataset by type. ....................................................... 28
3.1 SNIA Block I/O traces used in the paper ......................................................... 39
3.2 SNIA Trace idle interval duration analysis results ............................................. 41
3.3 Fixed waiting approach results for different disk traces ..................................... 56
4.1 The Duet API ........................................................................................................... 61
4.2 Event and State-based Notifications .................................................................. 61
4.3 Tasks adapted to use Duet and their characteristics .......................................... 65
4.4 Duet evaluation metrics ...................................................................................... 70
4.5 Maximum utilization with and without Duet for Btrfs maintenance tasks: The table shows the maximum utilization (in 10% intervals) at which each maintenance task can still complete its work in a 30-minute interval. Higher values indicate that the foreground workload is able to utilize the device more. Higher read-write ratios, higher data overlap and a uniform distribution of file accesses improve opportunistic processing, allowing higher maximum utilization. .............................................................................. 72
4.6 Segment cleaning time with and without Duet ............................................... 75
4.7 Maximum utilization with and without Duet when multiple Btrfs tasks are combined. Similar to Table 4.5, we show the maximum utilization (in 10% intervals) at which each combination of maintenance tasks can still complete their work in a 30-minute interval. Higher values indicate that the foreground workload is able to utilize the device more. We find that without Duet, even low-utilized devices can struggle to complete all maintenance work. ......................................................................................... 76
List of Figures

2.1 Monthly probability of LSEs as a function of temperature. ........................................... 9
2.2 Monthly probability of LSEs as a function of temperature variability, as measured by the
coefficient of variation. ........................................................................................................ 10
2.3 Quartiles for the number of LSEs occurring in drives that have exhibited at least one LSE,
as a function of the disk’s temperature. The numbers correspond to disk model 6. ....... 10
2.4 Monthly LSE probability as a function of temperature, for young and old drive age groups. 11
2.5 Monthly LSE probability as a function of temperature, for drives with high and low write
and read loads. .................................................................................................................... 12
2.6 Monthly probability of a disk failure for individual disk models, as a function of temperature. 12
2.7 Disk throughput under a synthetic random read and random write workload, respectively,
as a function of disk internal temperature. Results for sequential read and sequential write
workloads were comparable. ............................................................................................... 15
2.8 Throughput under two different I/O intensive workloads (Postmark, OLTP) as a function
of disk internal temperature. ............................................................................................ 16
2.9 Monthly job error rates in a 3-year time period. ............................................................... 20
2.10 CDFs for the job error rate of individual backup system types. ...................................... 21
2.11 CDFs for the number of unique job error codes for each type of backup system. .......... 22
2.12 Percentage of job errors due to the $N$ most frequent error codes for different backup
system types. ...................................................................................................................... 22
2.13 Breakdown of job errors based on their cause: configuration issues, system errors, or
informational messages. ................................................................................................. 24
2.14 Architecture of a modern backup domain. ..................................................................... 27
2.15 Distribution of the average scheduling frequency of different job types across backup
domains. Recovery operations are broken into two groups of domains with more, and less
than 5 recovery operations each. Despite being of similar size, the characteristics of each
group differ significantly. ................................................................................................. 29
2.16 Tukey boxplots (without outliers) that represent the average size of full backup jobs, for
different job scheduling frequencies. Means for each boxplot are also shown. Frequent full
backups seem to be associated with larger job sizes, suggesting that they may be preferred
as a response to high data churn. ...................................................................................... 30
2.17 Probability density function for scheduling policy jobs at a given hour of a given day of
the week. Policies tend to be configured using the default scheduling windows at 6pm and
12am, resulting in high system load during those hours. ................................................. 31
2.18 Probability of a policy job occurring on a given day of the week, based on its type.
Incremental backups tend to be scheduled to complement full backups, while users initiate
recovery operations more frequently at the beginning of the week.

2.19 Distribution of the average rate at which the number of clients changes, across all domains
in our dataset. On average, 93% of client population changes are attributed to the addition
of clients.

2.20 Distribution of the average number of clients involved in each change of a domain’s client
population, across all domains in our dataset. The 95% confidence intervals (C.I.) for
each domain’s average are also shown.

2.21 Distribution of the average job size of a given job type across backup domains, after the
data has been deduplicated at the client side. Incremental backups resemble full backups
in size. Snapshot operations are not included, as they do not incur data transfer.

2.22 Distributions of average number of files transferred per job, across different job types.
The trends are consistent with those for job sizes (Figure 2.21).

2.23 Distributions of the average daily deduplication ratio of different job types, across backup
domains. Incremental and full backups observe high deduplication ratios, while the
uniqueness of metadata backups (management operations) makes them harder to dedu-
plicate.

3.1 Request activity for four disks from both the HP Cello and the MSR Cambridge traces.

3.2 Periods detected for the busiest 63 disks using ANOVA.

3.3 What fraction of a disk’s idle time do the largest idle periods make up? Note that the x
axis is cut off at the 50th percentile.

3.4 Expected idle time remaining for the traces in Table 3.1.

3.5 1st percentile of idle time remaining for the traces in Table 3.1, excluding TPC-C.

3.6 Fraction of idle time remaining for the traces in Table 3.1.

3.7 Comparison of the Auto-regression and Waiting approaches with the optimal (Oracle).

3.8 Comparison of different Waiting variants. The fixed variant prevails.

3.9 Response times for different ATA VERIFY sizes.

3.10 Architecture of our kernel scrubber.

3.11 Comparison of our user- (U) and kernel-level (K) scrubbers. Numbers above the bars give
the scrubber throughput.

3.12 Service times for different SCSI VERIFY sizes.

3.13 Impact of scrubbing parameters on sequential and staggered scrubbing performance.

3.14 Throughput comparison of sequential and staggered scrubbing, when run alongside a
synthetic workload (64KB segment size, 128 regions).

3.15 Response Times CDFs during the replay of a real I/O trace.

4.1 File access distributions for Microsoft traces and the Filebench benchmark.

4.2 Filebench webserver latency is barely affected when maintenance runs with low priority
under CFQ.

4.3 I/O saved when the scrubbing task is run together with the webserver workload.

4.4 I/O saved when the backup task is run together with the webserver workload.

4.5 I/O saved when the defragmentation task is run together with the webserver workload.

xi
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.6</td>
<td>Runtime speedup with different overlap for Rsync</td>
<td>75</td>
</tr>
<tr>
<td>4.7</td>
<td>I/O saved when the scrubbing and backup tasks are run together with the webserver workload</td>
<td>77</td>
</tr>
<tr>
<td>4.8</td>
<td>Maintenance work completed when scrubbing and backup are run together with the webserver workload</td>
<td>77</td>
</tr>
<tr>
<td>4.9</td>
<td>I/O saved when scrubbing, backup, and defragmentation are run together with the webserver workload</td>
<td>77</td>
</tr>
<tr>
<td>4.10</td>
<td>Maintenance work completed when scrubbing, backup, and defragmentation are run together with the webserver workload. Without Duet, maintenance work fails to complete even when the device is idle.</td>
<td>78</td>
</tr>
<tr>
<td>4.11</td>
<td>CPU overhead of Duet</td>
<td>79</td>
</tr>
<tr>
<td>4.12</td>
<td>I/O saved on a solid-state drive</td>
<td>79</td>
</tr>
<tr>
<td>4.13</td>
<td>Percentage of HDFS blocks read from cache (hit rate) for the second of two identical jobs.</td>
<td>83</td>
</tr>
<tr>
<td>4.14</td>
<td>Total time taken to synchronize and chunk Seafile repositories of different sizes with, and without using Duet events.</td>
<td>86</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Modern enterprise environments impose a multitude of demands on storage systems, including performance scalability, high availability and data security. In addition, they require various storage-related capabilities for meeting service-level agreements and legal needs, such as data retention, disaster recovery, data mining and storage analytics. To meet these diverse demands, storage systems rely on various types of maintenance tasks. These tasks run in the background, helping improve storage reliability, performance, security, or enabling data analysis. Common reliability and security tasks are backup and archiving [20, 27, 51, 57, 71, 96, 104, 241, 260], data scrubbing [29, 72, 127, 163, 164], write verification [186], and scanning for viruses [24, 76, 106, 117, 157, 219]. Performance-related tasks include data layout optimization [99, 209, 215], garbage collection [247], and deduplication [69]. Analysis tasks include data mining [184] and storage data analytics [50].

In a way, maintenance work resembles a form of insurance for storage systems. Similar to insurance policy conditions, adjusting the type and frequency of maintenance entitles storage system operators to different guarantees, at the cost of disrupting operations of user applications. It is the responsibility of the operator to choose which configuration is worth the upfront cost of avoiding a future disaster. The work presented as part of this thesis focuses on three relevant goals. First, we outline the challenges of performing maintenance in the field, based on studies we have carried out [9, 10, 68]. Second, we examine ways of tuning maintenance tasks and scheduling their I/O operations, in order to lessen their impact on user applications [12]. Unfortunately, multiple maintenance tasks are deployed in storage systems today. The final contribution of this thesis is a framework that informs tasks of changes in the system cache [11]. Using this information, tasks can reduce their I/O by prioritizing processing of cached data. As a result, tasks running concurrently can reduce the I/O required for maintenance and by extension, reduce the idle time needed to execute it.

1.1 Storage maintenance in the field

To avoid the prohibitive cost incurred in the absence of maintenance, storage operators today perform maintenance operations periodically. To test the necessity and costs of periodic maintenance when reliability guarantees need to be provided, we studied data from Google’s data centers [68] and set out to answer the following questions:

- Are hard disk failure characteristics similar across models and operating environments?
• Do periodic data accesses contribute to a device’s wear-out, and subsequently to failures?

• What is the cost of protective mechanisms built into storage devices today?

Overall, we find that predicting hard disk failures is challenging, and recent research efforts that attempt to do so depend on hardware support for metrics that need to be updated through periodic maintenance tasks [145, 147]. Our analysis corroborates the importance of developing the efficiency of these techniques further, as we find that hard disk failure rates differ both across drive models, and across operating environments. Furthermore, we find that protective mechanisms currently built into these devices can significantly impact performance when triggered. Our results are encouraging for periodic maintenance, as we find no evidence that additional data accesses lead to higher failure rates.

An important challenge of maintenance is that it can significantly impact the performance of user applications by increasing the latency of operations being performed at the same time. To avoid interfering with user applications, maintenance work is usually scheduled during dedicated downtime periods, when normal system operation is suspended. However, provisioning sufficient downtime for maintenance is not trivial, and over-provisioning can be costly. Moreover, finding the least intrusive time to suspend system operation across all users can be challenging. This problem is exacerbated in shared environments, such as the cloud, where multiple time zones are at play. To better understand the problems that administrators face when operating maintenance software today, we conducted a study of enterprise backup systems that use Veritas backup software [10], which provides answers to the following questions:

• How often do backup operations fail?

• What are the most frequent failure modes for backup operations?

• Is the provisioned downtime sufficient for completing backup work?

Another challenge with periodic maintenance is that data accesses due to maintenance operations need to be performed multiple times, requiring more than the idle time available in the system, which makes it more challenging to mitigate their impact on normal system operations. To gauge the degree of this problem, we performed a study of backup systems deployed by Symantec’s enterprise customers [9], with a goal to answer the following questions:

• How often are backup operations performed?

• What is the amount of data accessed and transferred over the network during a typical backup?

Overall, we find that full backups are performed as often as every 1-4 days, in addition to daily maintenance jobs of other types. Despite their frequency, individual backup operations still transfer tens of gigabytes of data over the network, even with recorded deduplication ratios that are as high as 90% on average. The latter further implies that the data accessed during a backup are an order of magnitude larger, i.e., hundreds of gigabytes per backup operation.

1.2 Scheduling maintenance I/O

To lessen the impact of maintenance on user applications, maintenance I/O should be scheduled at times when storage devices are otherwise idle. In order to achieve this, device idleness must be accurately
predicted. To understand idleness, we performed a statistical analysis of publicly available traces of disk workloads, and applied the resulting observations towards the design of idle time predictors. Using the most accurate predictors, we describe an approach that maximizes maintenance throughput for a given workload, while meeting a predefined slowdown goal for I/O requests issued by user applications.

To assess the applicability of our scheduling approach, we apply it in the context of scrubbing, a maintenance task widely used to provide guarantees on the reliability of storage devices [29, 72, 127, 163, 164]. To achieve this, we implemented scrubbing algorithms in both the user-level, and within the Linux kernel. Using these implementations, we provide the first known evaluation of the performance impact of different scrubbing algorithms on real-world workloads with, and without the use of our scheduling approach. We show that our scheduling approach outperforms the default Linux I/O scheduler, the only one to allow for I/O prioritization. We further provide conclusive answers to the questions: when should maintenance I/O requests be issued, and at what size, in order to achieve the highest throughput for a given slowdown of the foreground workload, assuming maintenance I/O requests of a fixed size.

1.3 Opportunistic storage maintenance

In the absence of sufficient idle time, it can be challenging to schedule maintenance work without impacting user applications. This problem occurs as maintenance tasks typically access large amounts of data that does not easily fit in memory, they run frequently [9, 212], and multiple types of tasks can be scheduled in a given system. However, long idle times may not be available for this maintenance work, especially with increasing data storage needs. For instance, as enterprises are moving to the cloud, data sharing occurs across time zones, and much higher consolidation ratios are observed in storage systems. As a result, workloads are losing their traditional diurnal characteristics that guarantee predictable idle periods, making it harder to meet maintenance goals.

To mitigate the impact of maintenance in these scenarios, we propose a novel maintenance approach that prioritizes processing of data cached in memory. Data may be cached as a result of other maintenance tasks requesting it, or due to overlapping foreground I/O activity. This approach reduces the impact of maintenance in two ways. First, maintenance tasks can implicitly collaborate with each other. For example, during a full logical file system backup, data layout reorganization (such as defragmentation or garbage collection) can be performed with no additional reads. Second, data that has recently been accessed can be provided to a maintenance task. For example, a block modified by the workload can be used by an incremental backup task, avoiding an additional read. This I/O reduction helps maintenance tasks complete their work within their scheduled windows. We implement our approach as Duet, a framework that provides notifications about page-level events to maintenance tasks, such as a page being added or modified in memory. Tasks use these events as hints to opportunistically process cached data.

1.4 Thesis contributions

The contributions of this thesis can be summarized in the following points.

- We present a large-scale study on the effect of temperature on the reliability of hard disk drives [68]. Our work confirms the importance of periodic maintenance and invalidates assumptions that
maintenance can contribute to device wear-out [202]. We also use a controlled thermal chamber to evaluate the high cost of protective mechanisms employed in hard disks today to improve their reliability.

- We present the first, large-scale study that attempts to characterize the failure modes of enterprise backup system jobs [10]. Our work sheds light on the problems faced by the administrators of these systems, such as difficulty to arrive at a correct configuration, and provisioning too little downtime to complete all backup work.

- We present the largest study of enterprise backup system workloads to date [9]. This work characterizes the frequency and size of operations in these systems, as well as the efficiency of deduplication mechanisms.

- To mitigate the impact of maintenance operations on the system’s normal operation, we evaluate the efficiency of several I/O scheduling techniques that schedule maintenance requests only when storage devices are otherwise idle [12]. Our scheduling algorithm designs are based on the results of our study of 77 diverse disk workloads, which attempts to characterize device idleness.

- To allow maintenance work to complete in scenarios where numerous maintenance operations need to be performed, we have developed a framework that allows maintenance tasks to collaborate on overlapping work [11]. We are further extending this idea to distributed systems [59], and as an enhancement to existing notification frameworks.

1.5 Thesis organization

The following chapters are organized as follows. Chapter 2 outlines the importance and challenges of storage maintenance in the field today, leveraging results from studies we conducted using data from Google’s datacenters, and enterprise customers of Veritas and Symantec. Chapter 3 describes and evaluates mechanisms that can be used to limit the impact of maintenance I/O requests on the foreground workload. Chapter 4 presents a framework that allows concurrently running maintenance tasks to reduce their I/O footprint, by leveraging cached data. Chapter 5 outlines related work that is relevant to the concepts covered in the thesis, and Chapter 6 motivates and outlines avenues for future work. Finally, Chapter 7 summarizes our conclusions.
Chapter 2

Storage maintenance in the field

This chapter outlines the challenges that storage operators face as part of performing storage maintenance, as we have identified them through the study of real-world data from three business organizations. Storage operators today perform maintenance operations periodically, in order to avoid the prohibitive cost incurred in the absence of maintenance. To understand the necessity of periodic maintenance, and the costs associated with it, we conducted a study on the reliability of hard disks deployed in Google’s data centers [68]. In Section 2.1 we describe our results, which demonstrate the heterogeneity of hard disk failure characteristics across disk models and operating environments. We further dispel beliefs that periodic data accesses contribute to device wear-out. Finally, we evaluate the high cost of protective mechanisms built into storage devices today.

Unfortunately, maintenance operations can interfere with user applications and impact their performance. To avoid this interference, storage operators typically schedule maintenance during dedicated downtime periods. Provisioning sufficient downtime for maintenance is not trivial, however, and over-provisioning is costly. To better understand the problems that administrators face when operating maintenance software today, we performed a study of enterprise backup systems using Veritas backup software [10]. In Section 2.2 we present our findings, which show that backup operations fail often due to configuration issues, and one of the most common failure modes is due to administrators provisioning too little downtime for maintenance work.

Another challenge with periodic maintenance is that data accesses due to maintenance operations need to be repeated. This increases the amount of maintenance work that needs to be performed, and as a result, its impact on user applications. To gauge the degree to which this poses a problem for storage systems today, we studied the frequency of backup operations and the growth rates of their sizes [9]. Our study focuses on enterprise backup systems deployed by Symantec customers. In Section 2.3 we describe our findings which show that backup operations are performed much more frequently than previously thought, and they require significant data transfers and accesses, despite the high efficiency of deduplication techniques.

2.1 The importance of periodic maintenance

To be successful, maintenance operations must be completed before they incur significant business cost. In some cases, the time frame for the completion of maintenance work is dependent on factors external
to the system. For example an infected file must be scanned for viruses before it is opened by a user, a data block should be backed up before its contents get corrupted by an adversary, and a set of files might be required to be defragmented before they get accessed by a performance-sensitive application. Predicting this time frame accurately can be challenging, and incorrect predictions could potentially prove disastrous. In these cases, storage operators resort to periodic storage maintenance as the de-facto solution.

When the time frame for maintenance is dependent on factors internal to the system, such as a hardware failure, recent work shows that predictive approaches can be more promising [145, 147]. Specifically, hard disk drive failures are shown to be predictable through firmware counters recording the number of drive sectors that have been relocated after becoming inaccessible. In order for these counters to remain up-to-date, however, periodic maintenance is required to access all sectors and detect problematic ones. Furthermore, a Google study by Pinheiro et al. shows that these counters may be unreliable, due to lack of proper support from the firmware [174]. For these systems, periodic maintenance is the only choice against disaster.

To test the necessity and costs of periodic maintenance when reliability guarantees need to be provided, we studied data from Google’s data centers. Our work targets the following questions:

- Are hard disk failure characteristics similar across models and operating environments?
- Do periodic data accesses contribute to a device’s wear-out, and subsequently to failures?
- What is the cost of protective mechanisms built into storage devices today?

Our study focuses on characterizing the effect of temperature on the reliability and performance of hard disk drives. Overall, we find that hard disk failure rates can differ both across drive models, and across operating environments. Our results are encouraging for periodic maintenance, as we find that data access rates are not correlated to drive failure rates. Finally, we show that protective mechanisms currently built into hard disks can significantly impact performance when triggered. As a result, we believe that periodic storage maintenance will remain an indispensable tool in the arsenal of storage operators until predictive approaches become sufficiently portable and reliable to obsolete it.

### 2.1.1 Case study: Choosing between energy efficiency and reliability

Data centers have developed into major energy hogs. The world’s data centers are estimated to consume power equivalent to about seventeen 1,000 MW power plants, equaling more than 1% of total world electricity consumption, and to emit as much carbon dioxide as all of Argentina [116]. More than a third, sometimes up to one half of a data center’s electricity bill is made up by electricity for cooling [32, 130]. For instance, for a data center consisting of 30,000 square feet and consuming 10MW, the yearly cost of running the cooling infrastructure can reach up to $4-8 million [169].

Not surprisingly, a large body of research has been devoted to reducing cooling cost. Approaches that have been investigated include, for example, methods to minimize air flow inefficiencies [169, 218], load balancing and the incorporation of temperature awareness into workload placement in data centers [40, 173, 181, 207], and power reduction features in individual servers [78, 82].

Interestingly, one key aspect in the thermal management of a data center is still not very well understood: controlling the setpoint temperature at which to run a data center’s cooling system. Data centers typically operate in a temperature range between 20°C and 22°C, some are as cold as 13°C
Due to lack of scientific data, these values are often chosen based on equipment manufacturers’ (conservative) suggestions. Some estimate that increasing the setpoint temperature by just one degree can reduce energy consumption by 2 to 5 percent [41, 44]. Microsoft reports that raising the temperature by two to four degrees in one of its Silicon Valley data centers saved $250,000 in annual energy costs [156]. Google and Facebook have also been considering increasing the temperature in their data centers [156].

While increasing data center temperatures might seem like an easy way to save energy and reduce carbon emissions, it comes with some concerns, the most obvious being its impact on system reliability. Unfortunately, the details of how increased data center temperatures will affect hardware reliability are not well understood and existing evidence is contradicting. A recent study [218] indicated that in order to avoid thermal redlining, a typical server needs to have the air temperature at its front inlets be in the range of 20-30°C. Every 10°C increase over 21°C decreases the reliability of long-term electronics by 50% [172]. Other studies show that a 15°C rise increases hard disk drive failure rates by a factor of two [15, 48]. On the other hand, a recent Google study [174] suggests that lower temperatures are actually more detrimental to disk reliability than higher temperatures.

To provide a better understanding of the issues involved in raising data center temperatures we conducted a study of a large amount of field data from several organizations [68]. While we have examined the reliability characteristics of several data center components, in 2.1.2 we focus on hard disks, which are one of the most frequently replaced components in modern data centers [200, 201]. Specifically, we analyze data from Google, spanning tens of data centers and tens of thousands of drives. Our goal is to understand two common failure modes of hard disks: latent sector errors, and complete disk failures. Other possible concerns of increasing data center temperatures include the effect on server performance, as many servers employ techniques such as CPU and memory throttling, or hard disk data verification mechanisms when temperatures reach a critical threshold. We examine this effect for hard disks. Finally, in Section 2.1.3 we use the observations of our study to derive guidelines for controlling the temperature of hard disks, and insights on the role of periodic storage maintenance used to improve their reliability.

2.1.2 Temperature and hard disk drives

Temperature and reliability

Latent sector errors (LSEs) are a common hard disk failure mode, where individual sectors on a disk become inaccessible. The data stored on the affected sectors is lost, unless the system can use redundancy mechanisms to recover it. LSEs happen at a significant rate in the field [28, 174], with 3-4% of all drives reporting LSE occurrences at some point in their life, and are expected to become more common as disk capacities increase. While recent work [28] has studied the prevalence and some statistical properties of LSEs, there is no prior work on how temperature affects this important error condition.

Hard disk failures include any kind of disk problems that are considered serious enough to replace the disk in question, including a high rate of LSEs. Hard disk failures are a serious condition, since they create the potential for data loss and happen at a significant rate: typically 1-5% of drives in a data center need to be replaced in a given year [174, 201]. The only existing work that uses field data to derive trends relevant to the effect of temperature on hard disk failures is the work by Pinheiro et al. [174]. Surprisingly, they found a strong drop in disk failure rates with increasing temperature, except for very
high temperatures (above 45°C). This is in contrast with common reliability models, which estimate
disk failure rates to increase exponentially with temperature. Our goal is to revisit the question of how
temperature affects disk failure rates, and study the effect of utilization, differences between models and
data centers, and the age of a disk.

For our study, we have obtained data on hard disk replacements and LSEs from 19 different data
centers at Google, covering 5 different disk models. This data was collected from January 2007 to May
2009. For each disk we have data on the age of the disk, the average temperature, and the average
utilization over the observation period as it is reported by the drive’s S.M.A.R.T. system, and whether
the disk was replaced during the observation period. For 3 hard disk models that are deployed at 7
different data centers we also have monthly reports of the temperature variance in that month, the
count of latent sector errors, and the number of read and write operations during that month. Our
dataset provides all these metrics without identifying individual disk drives of the specified model. As a
result, we cannot analyze the behavior of individual drives over time. Our results focus on trends that are
characteristic of all, or individual models. We note that the time period represented in our dataset does
not overlap in time with the study of Pinheiro et al. [174]. However, while the time period is different, the
measurement methodology and infrastructure used to collect the data is the same. Table 2.1 summarizes
the information in our hard disk failure dataset, and Table 2.2 summarizes the information in our LSE
dataset.

<table>
<thead>
<tr>
<th>Model ID</th>
<th>Number of data centers</th>
<th>Number of disks</th>
<th>Disk Months</th>
<th>Monthly disk failure probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
<td>7972</td>
<td>173,945</td>
<td>0.28%</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>5906</td>
<td>143,456</td>
<td>0.23%</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>93498</td>
<td>752,579</td>
<td>0.04%</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>69421</td>
<td>829,859</td>
<td>0.11%</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>95226</td>
<td>2,953,123</td>
<td>0.12%</td>
</tr>
</tbody>
</table>

Table 2.1. Hard disk failure data used in our analysis, collected from Google data centers.

<table>
<thead>
<tr>
<th>Model ID</th>
<th>Number of data centers</th>
<th>Number of disks</th>
<th>Disk Months</th>
<th>Average monthly LSE probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>3</td>
<td>18,692</td>
<td>300,000</td>
<td>0.63%</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>17,515</td>
<td>300,000</td>
<td>1.77%</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>36,671</td>
<td>300,000</td>
<td>0.67%</td>
</tr>
</tbody>
</table>

Table 2.2. Latent sector error data used in our analysis, collected from Google data centers.

Figure 2.1 shows the monthly probability of a disk experiencing an LSE as a function of the average
temperature, for each one of the disk models in our dataset. The error bars in this, and following figures
represent the 95% confidence intervals for each value; larger bars for higher temperatures are due to
insufficient data. In a data center, there are many factors that can affect reliability: workload, humidity,
power spikes, and handling procedures, to name a few. Thus, we also break down our results for each
disk model by data center. To put those numbers into perspective, note that most hard disks are rated
for a maximum operating temperature of 50-60°C.

As one might expect, we observe a trend of increasing LSE rates as temperature rises. However,
the magnitude of increase is much smaller than what should be expected based on common models and estimates, in particular when isolating the instances of LSEs per disk model and data center. Statistical models considering the effect of temperature on hardware components usually assume an exponential increase in failures as a function of temperature, based on the Arrhenius equation [110]. As a result, they predict that failure rates will roughly double for every 10-15°C increase in temperature [15, 48, 218]. Visual inspection of our graphs shows that only 5 out of the 10 (disk model, data center) combinations exhibit a clear increase in errors as temperature increases: (disk model 3, data center 2), (disk model 4, data center 8), (disk model 4, data center 9), (disk model 6, data center 3), and (disk model 6, data center 5). We also observe that the increase in error rates tends to be linear, rather than exponential, except for very high temperatures (above 50°C).

To formalize our previous observation, we fitted two different models to the data. The first is a simple linear model, i.e., we try to model the error rate $y$ as a function of temperature $t$ as $y = a_1 + a_2 \cdot t$. The Arrhenius model, one of the most common models for effects of temperature on hardware reliability, is an exponential one. Therefore, we also fit an exponential model to our data, i.e., we model the failure rate $y$ as a function of temperature $t$ as follows: $y = a_1 \cdot e^{-a_2/t}$. The detailed results, including model parameter values and goodness-of-fit estimates, are presented in the full paper [68]. Overall, we find that the linear model provides a fit of comparable or even better accuracy, as measured by the Sum of Squared Errors.

**Observation 1:** Latent sector errors occur at much slower rates than those suggested by reliability models. Half of our (disk model, data center) pairs show no evidence of an increase, while for the others the increase is linear rather than exponential.

In addition to the average temperature that a drive is exposed to, another important factor is temperature variability, since large variations in temperature can negatively affect IT equipment. To study the impact of temperature variability on LSEs we plot the monthly LSE probabilities as a function of the coefficient of variation (CoV) \(^1\) in Figure 2.2. We chose the CoV, rather than variance or standard deviation, since it is normalized by the mean. A positive correlation between LSEs and temperature variance could just be due to the positive correlation between LSEs and mean temperature. The increas-

\(^1\)Recall that the coefficient of variation is defined as the standard deviation divided by the mean.
Figure 2.2. Monthly probability of LSEs as a function of temperature variability, as measured by the coefficient of variation.

Figure 2.3. Quartiles for the number of LSEs occurring in drives that have exhibited at least one LSE, as a function of the disk’s temperature. The numbers correspond to disk model 6.

Despite the error margins, Figure 2.2 shows an increase in LSE probabilities for models with higher temperature CoV. We have verified these trends by fitting a linear model to capture the relationship between LSEs and the CoV, and find a positive slope for all (disk model, data center) pairs. It is also worth noting that the error probabilities in Figure 2.2 are significantly higher than those reported as a function of the average operating temperature in Figure 2.1.

**Observation 2:** Temperature variability tends to have a more pronounced and consistent effect on LSE rates than the average temperature.

Our analysis so far has exclusively focused on the probability of a drive developing LSEs. Another interesting question is whether higher temperature leads to a higher number of LSEs, once a drive starts developing them. To answer this question Figure 2.3 shows for those disk months that have errors the 25th and 75th percentiles of the number of LSEs that occurred, as well as the median. We only include
Figure 2.4. Monthly LSE probability as a function of temperature, for young and old drive age groups.

results for disk model 6, because all other disk models exhibit comparable trends. We focus on the quartiles, rather than the median, since we find the median number of LSEs to be highly variable and hence easily biased by outliers. We observe that the line for all quartiles is flat, indicating that hotter drives with errors do not experience a higher frequency of errors than colder drives with errors.

**Observation 3:** Higher temperatures do not increase the expected number of LSEs once a drive develops them, possibly indicating that the mechanisms causing LSEs are the same under high or low temperatures.

We also note that the rate of LSEs for the same model can vary greatly across data centers. For example, the error rate for disk model 3 is significantly higher (more than 2x difference) for data center 2 than for the other data centers, and the error rates for model 6 are significantly higher for data center 0 than for other data centers (again, more than 2x difference). This brings up the question whether factors such as environmental conditions, age, or usage of a drive affect how it reacts to temperature. While we have no data on environmental factors, such as humidity or the quality of supplied power, we have information on the age of each drive and its utilization. We use this data to study the effect of drive age and utilization in Figures 2.4 and 2.5, respectively.

Our study of age and temperature in Figure 2.4 focuses on model 6, since the disks for this model span the widest range in age. We divide the drives into two groups, those that are less than 18 months old and those that are 18-36 months old, and plot LSE probabilities as a function of temperature separately for each group. We find that both lines show similar trends with no evidence that older drives are more sensitive to higher temperatures.

**Observation 4:** Within a range of 0-36 months, older drives are not more likely to develop LSEs under temperature than younger drives.

Figure 2.5 studies the effect of workload intensity. Figure 2.5a divides disks into two groups, one with high read utilization and one with low read utilization, and plots the LSE probabilities separately for the two groups. We measure read utilization by the number of read operations per month and assign a disk to the low read utilization group if the number of read operations is below the median for the dataset, and to the high read utilization group otherwise. Figure 2.5b performs the corresponding analysis for
write utilization. Results are shown only for model 6, but other models exhibit similar trends.

We find that drives with higher utilization are not more sensitive to higher temperatures. That is an interesting result beyond the study of temperature effects, as it has been an open question as to how workload intensity affects LSEs. Methods that are intended to protect against data loss due to LSEs, such as running a periodic “scrubber” that reads the entire disk to proactively detect LSEs, place additional load on a system, and a concern is that this added load might increase the rate of LSEs. Our results indicate that such worries are, likely, unfounded. To add statistical rigour to Observations 4 and 5, we performed an ANOVA test. The results indicate no correlation between LSEs and write utilization. There is evidence for a correlation with read utilization and age, however this is due to drives with lower read utilization and lower age experiencing slightly increased rates of LSEs.

Observation 5: High utilization does not increase LSE rates, regardless of temperature.

Finally, we have studied hard disk failure rates as a function of temperature. Figure 2.6 shows
the monthly failure rate for each of the five disk models, averaged across all data centers. With the exception of disk model 3, we observe increasing failure rates with rising temperature. However, we note that the increase in failures tends to increase linearly rather than exponentially with temperature increase, except for very high temperatures (above 50°C). We validate this observation by fitting a linear and an exponential model to the data, following the same methodology we described for LSEs. We find that the slope of the curves tends to change for very high temperatures, so we also repeated the analysis by including only data points below 50°C. In all cases, the linear model provides a significantly better fit than the exponential model.

Observation 6: For temperatures below 50°C, disk failure rates grow more slowly with temperature than common models predict. The increase tends to be linear rather than exponential.

We also note that, unlike the Google study by Pinheiro et al. [174], we do not see a general trend for higher failure rates at lower temperatures. For example, the Google study reports more than 50% drop in failure rate when moving from 25-35°C. We believe that the reason for this, is the aggregation of data for different models and data centers in the same curve in the Google study. Different drive models operate at different temperatures (due to differences in their design), and individual drive models can also exhibit significant variations in their failure rates. As a result, it is possible that the data points at the lower end of the temperature spectrum contain more drives of a model that happened to run colder and have higher failure rates, hence biasing the results.

As was the case for LSEs, we find that for the same model, the monthly failure probabilities can vary greatly across data centers, even for the same temperature \(^2\). This points to other factors, beyond temperature, that have an equally strong or stronger effect on disk lifetimes and motivates us to study two possible factors that we have data on: age and utilization. We followed the same methodology as for LSEs, and divided the drives for each model into those with high and low read utilization, high and low write utilization, and based on age. We found that the behavior of a drive under higher temperatures did not change due to the degree to which it was utilized, or its age. We note that as before, we only have statistically significant data for disk ages up to 36 months.

Observation 7: Neither the utilization, nor the age of a drive have a significant effect on its failure rate as a function of the temperature it operates in.

Temperature and disk performance

Beyond potentially affecting server reliability, there are other concerns with raising data center temperatures. While it is widely known that higher temperatures might negatively affect the reliability and lifetime of hardware devices, less attention is paid to the fact that high temperatures can also negatively affect the performance of systems. For example, in order to protect themselves against a possibly increasing rate of LSEs, some hard disk models enable Read-after-Write (RaW) when a certain temperature threshold is reached. Under RaW, every write to the disk is converted to a Write Verify command, or a Write followed by a Verify operation, reading the sector that has just been written and verifying its contents [222, 223]. Unfortunately, features such as RaW are often considered trade secrets and are not well documented. In fact, even within a company manufacturing hardware those features and associated parameters are regarded confidential and not shared outside product groups. Our goal is to investigate

\(^2\)Full per-data-center graphs for disk failures are included in our tech-report [67].
Chapter 2. Storage maintenance in the field

experimentally how the performance of different hard disk models changes with increasing temperatures.

To study the performance of a server under increasing ambient temperatures, we set up a testbed using a thermal chamber. The thermal chamber is large enough to fit an entire server inside it, and allows us to exactly control temperature within a range of \(-10^\circ\text{C}\) to \(60^\circ\text{C}\) with a precision of \(0.1^\circ\text{C}\). How ambient temperature affects the temperature of server-internal components depends on many factors, including the design of the cooling system and the server and rack architecture. Therefore, instead of directly predicting the impact of data center ambient temperature on a system, we present our results as a function of the temperature of server internal components.

The server we use in our study is a Dell PowerEdge R710, a model that is commonly used in data center server racks. The server has a quad-core 2.26 GHz Intel Xeon 5520 with 8MB L3, with 16GB DDR3 ECC memory, running Ubuntu 10.04 Server with the 2.6.32 Linux kernel. We also equipped the server with a variety of different hard disk drives, including both SAS and SATA drives and covering all major manufacturers. The specifications of our hard disk models are shown in Table 2.3.

<table>
<thead>
<tr>
<th>Manufacturer</th>
<th>Model</th>
<th>Interface</th>
<th>Capacity</th>
<th>RPM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hitachi</td>
<td>Deskstar</td>
<td>SATA</td>
<td>750GB</td>
<td>7200</td>
</tr>
<tr>
<td>Western Digital</td>
<td>Caviar</td>
<td>SATA</td>
<td>160GB</td>
<td>7200</td>
</tr>
<tr>
<td>Seagate</td>
<td>Barracuda</td>
<td>SATA</td>
<td>1TB</td>
<td>7200</td>
</tr>
<tr>
<td>Seagate</td>
<td>Constellation</td>
<td>SAS</td>
<td>500GB</td>
<td>7200</td>
</tr>
<tr>
<td>Seagate</td>
<td>Cheetah</td>
<td>SAS</td>
<td>73GB</td>
<td>15000</td>
</tr>
<tr>
<td>Fujitsu</td>
<td>MAX3073RC</td>
<td>SAS</td>
<td>73GB</td>
<td>15000</td>
</tr>
<tr>
<td>Hitachi</td>
<td>Ultrastar</td>
<td>SAS</td>
<td>300GB</td>
<td>15000</td>
</tr>
</tbody>
</table>

Table 2.3. Specifications of the hard disk models used in our experiments.

We use a wide range of workloads in our experiments, including a set of synthetic microbenchmarks, and a set of macrobenchmarks aiming to model a number of real world applications:

**Random** A synthetic workload comprised of independent 64KB read (or write) requests issued back-to-back at random disk sectors.

**Sequential** Since a pure sequential workload would stress the on-disk cache, we opt for a synthetic workload with a high degree of sequentiality, instead. We pick a random disk sector, and issue back-to-back 64KB read (or write) requests on consecutive sectors for 8MB following the initial request.

**OLTP** Models TPC-C [229], a commonly used database benchmark modeling on-line transaction processing (OLTP). To make the workload I/O-bound, we configured the database with 70 warehouses (7GB), using 4GB RAM. We configured our DBMS so that logging and data were stored on separate disks, and thus the throughput reported is relevant to the disk’s transfer rate alone.

**DSS** We configured TPC-H [230], a commonly used database benchmark modeling decision support workloads (DSS), with a MySQL InnoDB database of 10GB and a 3.4GB buffer pool, resulting in a disk-bound workload.
To study the effect of temperature on disk performance, we ran our workloads on each of the drives in our testbed, while placing the drive in the heat chamber and gradually increasing the temperature inside the chamber. The two graphs in Figure 2.7 show the results for the random read and random write microbenchmarks, as a function of the drive internal temperatures, as reported by the drives’ S.M.A.R.T. interface. Results for sequential read and sequential write were similar and are, therefore, omitted. We observe that all SAS drives and one SATA drive (the Hitachi Deskstar) experience some drop in throughput for high temperatures. The drop in throughput is usually in the 5-10% range, but can be as high as 30%. Because of the fact that the throughput drop for a drive happens consistently at the same temperature, rather than randomly or gradually, and that none of the drives reported any errors, we speculate that it is due to protective mechanisms enabled by the drive. For example, in the case of the write workloads (which show a more significant drop in throughput) this drop in throughput might be due to the enabling of features such as RaW.

Next, we attempt to experimentally determine the temperature where throughput start to drop. We observe in Figure 2.7 drops at either around 50°C (for the Seagate SAS drives) or 60°C (for the Fujitsu and Hitachi SAS drives), which coincides with the maximum operating temperatures at which hard disks are typically rated (50-60°C). However, these are disk internal temperatures. When translating them to ambient temperatures (inside the heat chamber) we observe a drop in throughput for temperatures as low as 40°C (for the Seagate 73GB and Hitachi SAS drives), 45°C for the Fujitsu and Seagate 500GB SAS drives, and 55°C for the Hitachi Deskstar, ranges that are significantly lower than the maximum of 50-60°C that manufacturers typically rate hard disks for. While data centers will rarely run at an average inlet temperature of 40°C or above, most data centers have hot spots [68], which are significantly hotter than the rest of the data center, and which might routinely reach such temperatures. Specifically, in most cases the temperatures where throughput drops start to occur translate to ambient temperatures...
in the 40°C range. For the Hitachi and Fujitsu SAS drives throughput drops start as early as 30°C ambient temperature.

Figure 2.8 shows how temperature affects the throughput of two of our macrobenchmarks, Postmark and OLTP. We observe similar trends as for the microbenchmarks, with throughput drops at the same temperature point. Interestingly, the order of magnitude in the throughput drop for Postmark and OLTP is in most cases even larger than for the synthetic microbenchmarks. The drop in throughput for the 300GB Hitachi and 73GB Seagate increases to 10-20%, while the throughput drop for Fujitsu and 500GB Seagate drives is in the range of 40% and 80%, respectively. The drops observed for DSS-disk looked more similar in magnitude to those for the synthetic benchmarks.

2.1.3 Implications on maintenance frequency

Based on our study of data spanning more than a dozen data centers at Google, we find that the effect of high data center temperatures on hard disk reliability are smaller than often assumed. Specifically, the correlation between latent sector errors or whole-disk failures and temperature is much weaker than expected. We find that for (internal) device temperatures below 50°C, errors tend to grow linearly with temperature, rather than exponentially, as existing models suggest.

It is important to note that this does not mean that high temperatures have no effect on hardware reliability, or that the Arrhenius model is flawed. But it might mean that the effects of other factors dominate failure rates. The Arrhenius model tries to solely capture the effect of heat on hardware components without taking into account other possible factors that impact hardware reliability in the field. Anecdotal evidence from discussions with data center operators suggests, for example, that poor handling procedures for equipment are a major factor in the field (which is hard to capture in measurement data). Our results indicate that, all things considered, the effect of temperature on hardware reliability is actually weaker than commonly thought.

We also find that, rather than average temperature, the variability in hard disk temperature might be the more important factor. Efforts in controlling such factors might be more important in keeping
hardware failure rates low, than keeping temperatures low. On the other hand, we find that factors expected to affect disk failure rates, such as the age of a drive, show weak correlations as temperatures increase. More importantly, we find that hard disk failure rates vary significantly across disk models, and likely due to the above environmental factors failure rates differ even when the same model is deployed in different data centers. Therefore, until we further develop our understanding of factors that affect the reliability of hard drives, our work suggests that periodic maintenance might be a safe alternative to guarantee data reliability.

One concern with periodic maintenance in the case of reliability, is whether it adversely affects the guarantees it provides. Specifically, we find that the error mode that is most strongly correlated with high temperatures are LSEs. A common method for protecting against data loss due to LSEs include Read-after Write (RaW) and periodic “scrubbing” of the hard disk to proactively detect such errors. We have performed experiments using a testbed based on a thermal chamber, and we observed evidence that (enterprise class) hard disks do employ mechanisms, such as RaW, but we find that they tend to kick in only at very high temperatures which differ across drives and are associated with significant performance penalties. Our setup and results are included in our paper [68], and are explained in further detail in our technical report [67]. On the other hand, we find that concerns about periodic scrubbing are likely unfounded, which makes this type of maintenance the better approach to defend against LSEs. Specifically, some fear that the extra workload placed on a drive by the scrubber task might lead to early wear-out of the drive [202], but we see no correlation between a drive’s workload intensity and it’s failure probability.

2.2 Finding time for maintenance

An important concern when performing storage maintenance is that it can interfere with the system’s normal operation. To avoid interfering with user applications, maintenance today is typically scheduled during dedicated downtime periods. Downtime is usually scheduled by the system’s administrator, and during that period all system operations are suspended.

A key downside of maintenance downtime is business cost. Unfortunately, the exponential data growth rates witnessed by enterprises today [208], have not been matched by corresponding increases in hard disk performance, where this data is primarily stored [42]. As a result, administrators are faced with the conundrum of taking the system offline for longer periods, or skipping maintenance work and risking a violation of the service-level agreements provided by the maintenance tasks to data consumers.

Another significant downside of maintenance downtime is that it is becoming increasingly harder to schedule. As businesses move towards shared infrastructures, such as the cloud, data is accessed by users in multiple time zones. Consequently, finding the right time to take the system offline without affecting the users turns into a non-trivial problem.

To better understand the problems that administrators face when operating maintenance software today, we conducted a study of enterprise backup systems using Veritas backup software [10]. The goal of this work is to find answers to the following questions:

- How often do backup operations fail?
- What are the most frequent failure modes for backup operations?
- Is the provisioned downtime sufficient for completing backup work?
Our study confirms that maintenance poses several challenges to administrators. We find job errors to be frequent, and confirm that trends reported in the software reliability literature also hold for backup systems, such as that the majority of job errors are due to configuration issues. We further discover that one of the common errors for backup system jobs is that too little downtime is available for them to complete.

### 2.2.1 Case study: Understanding how enterprise backups fail

From enterprise organizations to home users, data backups are still the preferred mechanism used to prevent data loss in the event of catastrophic failures. The guarantees provided by backup software, however, rely on the assumption that periodic backup jobs will complete successfully. Unfortunately, backup software is becoming increasingly complex to configure and manage, as more tuning parameters are introduced to handle diverse application requirements. Recent surveys of CIOs and IT professionals show that 27% of businesses have trouble recovering from backups due to backup jobs not completing successfully [108], and 80% experience challenges managing backup data and configuring the backup software [237]. At the same time, data generation rates increase exponentially [208], leaving less time to repeat failed jobs.

The goal of this analysis is to help researchers and practitioners understand how backup system jobs fail, and identify the factors that can be used to predict these failures. Our results are based on the analysis of periodic reports collected from customer installations of Veritas NetBackup [241], a commercial backup software product, over the span of 3 years. In total, we studied 775 million jobs from more than 20,000 backup systems, i.e. customer installations. Depending on their type, these jobs perform specific operations, such as data backup, recovery, and backup data management (e.g. replication). Jobs are scheduled according to backup policies, which are sets of configurable parameters dictating how individual applications should be backed up. For example, VMware policies expose parameters specific to virtual machines, while Oracle policies expose parameters relevant to database instances. Our dataset is explained in detail in Section 2.2.2.

In Section 2.2.3, we investigate the prevalence of errors in backup systems and their causes. Table 2.4 summarizes the most important observations in our analysis. We find that backup system jobs fail frequently, and the majority of errors are attributed to configuration issues, which confirms a recurrent trend in the literature for other system types [89, 159, 162, 224, 254]. On the bright side, we find that
Chapter 2. Storage maintenance in the field

<table>
<thead>
<tr>
<th>Job type</th>
<th>Number</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Backup jobs</td>
<td>604.9 Million</td>
<td>Create backup images</td>
</tr>
<tr>
<td>Management jobs</td>
<td>105.8 Million</td>
<td>Manage (e.g. delete, replicate, migrate) backup images</td>
</tr>
<tr>
<td>Snapshots</td>
<td>58.2 Million</td>
<td>Create Copy-on-Write data snapshots</td>
</tr>
<tr>
<td>Recovery jobs</td>
<td>6.3 Million</td>
<td>Recover data from backup images</td>
</tr>
<tr>
<td><strong>Total jobs</strong></td>
<td><strong>775.2 Million</strong></td>
<td></td>
</tr>
</tbody>
</table>

Table 2.5. Breakdown of job types in our dataset.

the errors themselves are not diverse, and the 10 most frequent error codes returned by jobs account for more than 78% of job errors. We explain these errors in detail, so that designers can take heed in order to improve the robustness of future backup software.

2.2.2 Veritas dataset description

Our analysis and models are based on data from telemetry reports collected from enterprise customer installations of a commercial backup product, Veritas NetBackup [241]. Reports are only collected from customers who opt to participate in the telemetry program, and contain no personal identifiable information. This section outlines the characteristics of our telemetry dataset.

**Reported information.** Telemetry reports received from customer backup systems contain runtime and configuration information about these systems. This runtime information describes the jobs that run in each system, and whether they completed successfully. The different job types supported by monitored backup systems are described in Table 2.5. When a job fails to complete successfully, we collect its error code. Our study uses these job errors, in combination with other system or job characteristics, to investigate why jobs fail.

**Dataset size and breadth.** Our dataset consists of 1 million weekly telemetry reports, collected over the span of three years, between September 2012 and August 2015. These reports represent 22 minor versions of the backup software, all under the same major version: NetBackup 7.6. More than 20,000 customer installations are represented, across most modern operating systems.

**Monitoring duration.** The backup systems described in our study were each monitored for 5.7 months on average, and a maximum of 34 months. As part of data pre-processing, we have excluded almost 5,000 systems that were monitored for less than one week, or used internally at Veritas. Note that the monitoring time is not always equivalent to the total lifetime of the backup system, as most of these systems were still online at the time we conducted our analysis.

2.2.3 Error characteristics

In this section, we use our dataset to characterize the frequency of job errors in backup systems. We further investigate the diversity in the error codes returned by individual jobs, and analyze them to find the most popular causes for job errors.
Prevalence of job errors

After pre-processing, our dataset consists of 775.2 million jobs across 15,503 systems. Of these jobs, 69.4 million (or 8.7%) terminate with an error code indicating a partial, or complete failure. Only 3,243 systems, or 20.9%, exhibit no errors during their lifetime. Surprisingly, almost all of these systems are short-lived with only a few jobs, often used for testing the software.

In Figure 2.9, we show the monthly job error rate across all jobs. Note that as more systems join the telemetry program (right y-axis), the error rate becomes more stable (left y-axis). Another reason for the significant drop in the error rate during 2014 is the general availability of the first stable release of the version of NetBackup for which we are collecting telemetry. The systems contributing telemetry reports prior to this release date are mostly testing and development systems, which we describe in more detail later. However, error rates remain high even after the drop. As a result, we conclude that a better understanding of backup system errors is pertinent.

It is common for customers to test the backup software before deployment in the field. The installations used in this process have radically different workload and error profiles, as will be demonstrated later in this section. Thus, we have labeled individual systems and report results on each category separately. The different system categories are listed in Table 2.6. Development systems are run by partner organizations that have early access to new software features through alpha and beta versions. These systems are short-lived, but run a relatively large number of jobs. Test systems, on the other hand, are customer systems that run stable versions of the software, testing it before release in the field. They are
also short-lived, but considerably less busy relative to development systems, running an average of 112 jobs over 10 days of operation. The remaining installations correspond to production systems, which are by far the busiest and most long-lived systems.

The three system categories also differ with regards to the job error rate exhibited per system. Figure 2.10 shows the empirical cumulative distribution functions (CDF) for the system job error rates of individual system types. Development and test systems exhibit job errors 48.5% and 50.9% of the time on average, respectively, and their job error rates approximate the uniform distribution. This is surprising, because production systems follow a radically different distribution of job error rates. Through Kolmogorov-Smirnov tests [114], we confirmed that the job error rates for production systems follow a Weibull distribution, with a shape parameter of \(76.1 \pm 0.9\) × 10\(^{-2}\) and a scale parameter of 13.0 ± 0.3. Note that the Weibull distribution is typical in systems reliability research, especially for modeling failure rates [182, ch.2.12]. The error rate for jobs in production systems is 15.2% on average. This number differs from the numbers in Figure 2.9, which aggregates jobs across all backup systems.

**Observation 8:** 15.2% of jobs terminate with an error in production backup systems. Testing and development systems exhibit up to 3.3 times more errors on average.

**Error diversity**

In NetBackup, there are 1,194 distinct error codes that can be returned when a job fails partially, or completely [240]. Of these error codes, only 333 (or 27.9%) are reported in our dataset.

Backup systems of different types exhibit different sets of error codes. Specifically, 21 error codes occur only in development systems. These errors are caused by commands failing due to permission or communication issues, invalid inputs, and unsupported or unavailable software features. Similarly, 59 error codes are specific to production systems. These errors occur as a result of communication errors due to offline system components, product licensing issues, or configuration issues, such as the inability to complete all scheduled jobs because the downtime period defined by the administrator was too short. Interestingly, the set of error codes that occur in test systems is a strict subset of the errors occurring in both development and production systems. In other words, error codes that occur during testing are
likely those that survived development, but they make up only a strict subset of the potential errors in a production system.

**Observation 9:** 6% of error codes show up only in development systems. Production systems exhibit an order of magnitude more error codes than test systems.

The set of exhibited error codes can also differ across individual backup systems. Figure 2.11 shows the empirical CDFs for the number of unique error codes that individual backup systems exhibit throughout their lifespan. While average test and development systems display 2.4 and 4.9 error codes respectively, production systems can exhibit 22.3 error codes, almost an order of magnitude difference. Moreover, we find that the number of unique error code counts is positively correlated with the lifespan of the system (linearly, with a Pearson’s coefficient of 0.4), the number of job errors (non-linearly, with a Spearman’s coefficient of 0.6), and the number of jobs started (non-linearly, with a Spearman’s coefficient of 0.7). We analyze these correlations further in the full paper [10].

The number of error codes exhibited by a system is not an indicator of the effect that each error code
has on the system’s job error rate. We find that a few error codes are responsible for the vast majority of job errors. In Figure 2.12, we show the percentage of job errors that correspond to the $N$ most frequent error codes. Jobs are partitioned by system type. Overall, we find that the 10 most frequent error codes correspond to 78.1%, 79.0%, and 89.5% of job errors in production, development, and test systems respectively. Among these errors, half are unique for each system type, while the remaining 5 error codes are common across all systems. Of those, the most common error code denotes a partially successful backup job, where backing up of some data failed because the backup process was unable to access it.

Other common error codes occur due to insufficient backup storage space, short maintenance downtime windows that resulted in job abortion, or because the storage unit was found offline. Specifically, 1 in 6 failed jobs are aborted due to insufficient downtime, highlighting the difficulty of finding enough time for maintenance. To alleviate this problem, we discuss how maintenance work can be scheduled at times when the system is idle in Chapter 3, and propose an approach that allows maintenance I/O to be reduced in Chapter 4.

**Observation 10:** The 10 most frequent error codes correspond to 78.1-89.5% of job errors, depending on the system type. Backups often fail because of unavailable or insufficient resources (disk space, or short downtime windows).

**Error causes**

A smaller number of errors in a backup system implies fewer failure scenarios that need to be accounted for. It is not indicative, however, of the effort necessary to resolve each failure. To estimate the degree of effort required, we categorize error codes based on their cause, and analyze the distribution of job error causes.

We manually categorized the error codes that appear in our dataset, using the troubleshooting guide provided to administrators of the backup software [240], into three categories. **Configuration issues** are defined as those that can be attributed to parameters that have been assigned incorrect values, or system components that have been configured in a way that is incompatible with the operation of the backup system. Examples of such errors include: failed jobs due to incorrect file permissions or locked files, backups that failed to get scheduled because the specified window for scheduling was too short, authentication errors, and jobs scheduled in a way that caused the system to run over its storage capacity. **System errors** are defined as errors that may originate at the software or hardware layer, and describe scenarios that are not directly in the control of the system’s administrator. Examples of such errors include: unavailability of system components due to issues at the storage layer, errors at the operating system level, and internal errors of the backup software or applications it works with. Finally, **informational messages** are error codes describing non-fatal but unusual scenarios, such as jobs terminated by the administrator, and product licensing issues.

In Figure 2.13, we show a breakdown of job errors based on their cause. A separate breakdown is provided for each system type. As is evident, the majority of job errors are due to configurations issues: 90% in test systems, and over 75% for development and production systems. 6 out 10 most frequent error codes fall under this category, and account for more than 66% of job errors across all system types. The remainder of the errors are mostly due to system errors, while 1.4-2.4% are informational messages. This result is encouraging because configuration issues usually identify which parameters were incorrectly set, or the constraints that did not apply, causing the error. As a result, they are significantly easier to
address compared to system errors, which could start at the hardware level, or any of the storage and operating system layers above. Therefore, we believe that these configuration issues lend themselves to self-healing techniques.

**Observation 11:** *Depending on system type, 75-90% of job errors are due to misconfigurations.*

### 2.3 Too much work, too little time

A challenge introduced with periodic maintenance is that data accesses due to maintenance operations need to be performed multiple times. This makes it more challenging to mitigate the impact of maintenance on normal system operations, especially as idle time becomes insufficient. To gauge the degree of this problem, we performed a study of backup systems deployed by Symantec’s enterprise customers [9]. The goal of our study is to answer the following questions:

- How often are backup operations performed?
- What is the amount of data accessed and transferred over the network during a typical backup?

Overall, we find that full backups are performed frequently, as often as every 1-4 days, in addition to daily maintenance jobs of other types. Despite their frequency and deduplication ratios that are as high as 90% on average, individual backup operations still transfer tens of gigabytes of data over the network. The latter further implies that the data accessed during a backup are an order of magnitude more, i.e., hundreds of gigabytes per backup operation. Our observations suggest that periodic maintenance creates a significant amount of work for the storage system. This problem is further exacerbated for systems that rely on multiple maintenance tasks, and in the following chapters we will examine approaches to alleviate the burden of maintenance on these systems.

---

**Figure 2.13.** Breakdown of job errors based on their cause: configuration issues, system errors, or informational messages.
2.3.1 Case study: Characterizing enterprise backup system workloads

Studies analyzing the characteristics of storage systems are an important aid in the design and implementation of techniques that can improve the performance and robustness of these systems. In the past 30 years, numerous file system studies have investigated different aspects of desktop and enterprise systems [4, 30, 33, 63, 98, 134, 150, 165, 192, 197]. However, little work has been published to provide insight in the characteristics of backup systems, focusing only on deduplication rates [168], and the characteristics of the file systems storing the backup images [243]. With this study, we look into the backup application generating these images, their internal structure, and the characteristics of the jobs that created them.

Modern data growth rates and shorter recovery windows are driving the need for innovation in the area of data protection. Recent surveys of CIOs and IT professionals indicate that 90% of businesses use more than two backup products [61], and only 28% of backup jobs complete within their scheduled window [10, 108, 237]. The goal of this study is to investigate how data protection systems are configured and operate. Our analysis shows that the inefficiency of backup systems is largely attributed to misconfigurations. We believe that automating configuration management can help alleviate these configuration issues significantly. Our findings motivate and support research on automated data protection [73, 95], by identifying trends in data protection systems, and related directions for future research.

Our study is based on a million weekly reports collected in a span of three years, from 40,000 enterprise backup systems, also referred to as domains in the rest of the section. Each domain is a multi-tiered network of backup servers deploying Symantec NetBackup [221], an enterprise backup product. To the best of our knowledge, this dataset is the largest in existing literature in terms of both the number of domains, and the time span covered. Using this data, we are able to analyze the characteristics of a diverse domain population, and its evolution over time.

The configuration of a backup domain, with regards to job frequency and scheduling, is an important contributor to the overall system’s resource consumption. Understanding common practices employed by systems in the field can give us better insight in the load that these systems face, and the characteristics of that load. To derive these trends, we analyzed 210 million jobs performing a variety of tasks, ranging from data backup and recovery, to management of backup archives. We find that jobs occur in bursts, due to the preference of default scheduling parameters by users. Moreover, job types are strongly correlated to specific days and times of the week. To avoid these bursts of activity, we expect future backup systems to follow more flexible scheduling plans based on data protection guarantees and resource availability [26, 84, 151].

We also investigate how backup domains are configured. Identifying common growth trends is useful for provisioning system resources, such as network or storage bandwidth, to accommodate future growth. We find that the population of protected client machines grows in bursts and rarely shrinks. Overall, our findings suggest that automated configuration is an important and feasible direction for future research to accommodate growth bursts in the number of protected clients. Furthermore, successful resource provisioning for backup storage capacity requires data growth rate knowledge. Our results show that jobs in the order of tens of GBs are the norm, even with deduplication ratios of 88%. Also, retention periods for these jobs are selected as a function of backup frequency, and backups are performed at intervals significantly shorter than the periods for which they are retained. Thus, future data protection offering faster backup and recovery times through the use of snapshots [2, 73], will have to be designed to handle significant data churn, or employ these mechanisms selectively.

We summarize the most important observations of our study in Table 2.7. The rest of the section is


<table>
<thead>
<tr>
<th>Backup system characteristic</th>
<th>Observation</th>
<th>Previous work</th>
</tr>
</thead>
<tbody>
<tr>
<td>Job frequency</td>
<td>Full backups tend to occur every few days, while incremental ones occur daily. Recovery operations occur for few domains, on a weekly or monthly basis.</td>
<td>None</td>
</tr>
<tr>
<td></td>
<td>Users prefer default scheduling windows during weekdays, resulting in nightly bursts of activity.</td>
<td></td>
</tr>
<tr>
<td>Protected clients</td>
<td>Clients tend to be added to a domain in groups, on a monthly basis.</td>
<td>None</td>
</tr>
<tr>
<td>Job sizes</td>
<td>Incremental and full backups tend to be similar to each other in terms of size and number of files. Recovery jobs restore either few files and bytes, or entire volumes.</td>
<td>Considers file sizes instead [243]</td>
</tr>
<tr>
<td>Deduplication ratios</td>
<td>Deduplication can result in the reduction of backup image sizes by more than 88%, despite average job sizes ranging in the tens of gigabytes.</td>
<td>We confirm their findings [243]</td>
</tr>
</tbody>
</table>

Table 2.7. Summary of our most important observations regarding backup system workloads.

organized as follows. In Section 2.3.2, we provide some background on the anatomy of modern backup systems. Section 2.3.3 describes the dataset used in this study. Sections 2.3.4 and 2.3.5 present our analysis results on job scheduling and data growth, respectively.

2.3.2 Anatomy of modern backup systems

Modern backup domains typically consist of three tiers of operation: a master server, one or more storage servers, and several clients, as shown in Figure 2.14a. The domain’s master server maintains information on backup images and backup policies. It is also responsible for scheduling and monitoring backup jobs, and assigning them to storage servers. Storage servers manage storage media, such as tapes and hard drives, used to archive backup images. By abstracting storage media management in this way, clients can send data directly to their corresponding storage server, avoiding a bandwidth bottleneck at the master server. Finally, domain clients can be desktops, servers, or virtual machines generating data that is protected by the backup system against failures. In an alternative 2-tiered architecture model (Figure 2.14b), the storage servers are absent and the storage media are directly managed by the master server. The majority of enterprise backup software today, including Symantec NetBackup, support the 3-tiered model [20, 27, 51, 57, 71, 96, 104, 241, 260].

Performing a backup generally consists of a sequence of operations, each of which is executed as an independent job. Such jobs include: snapshots of the state of data at a given point in time, copying data into a backup image as part of a full backup, copying modified data since the last backup as part of an incremental backup, restoring data from a backup image as part of a recovery operation, and managing backup images or backing up the domain’s configuration as part of a management operation. These jobs are typically employed in a predefined order. For example, a full backup may be followed by a management operation that deletes backup images past their retention periods.

To be consistently backed up, or provide point-in-time recovery guarantees, business applications may require specific operations to take place. In these scenarios, backup products offer predefined policies that are specific to individual applications. For instance, a Microsoft Exchange Server policy will also backup the transaction log, to capture any updates since the backup was initiated. Users can further
configure policies to specify the characteristics of backups jobs, such as their frequency and retention rate.

2.3.3 Symantec dataset description

Similar to the work presented in Section 2.2, our analysis is based on telemetry reports collected from customer installations of a commercial backup product, Symantec NetBackup [221], in enterprise and regular production environments. The reports contain no personal identifiable information, or details about the data being backed up.

**Report types.** Each report in our dataset belongs to exactly one of three types: installation, runtime, or domain report. Reports of different types are collected at distinct points in the lifetime of a backup domain. *Installation reports* are generated when the backup software is successfully installed on a server, and can be used to determine the time each server of a domain first came online. *Runtime reports* are generated and transmitted on a weekly basis from online domains, and contain daily aggregate data about the backup jobs running on the system. *Domain reports* are also generated and transmitted on a weekly basis, and report daily aggregate metrics that describe the configuration of the backup domain. The latter two report types collect data similar to what is described in Section 2.2. The telemetry report metrics used in our analysis are summarized in Table 2.8.

**Dataset size.** The telemetry reports in our dataset were collected over the span of 3 years (January 2012 to December 2014). We collected 1 million reports from over 40,000 server installations deployed in 124 countries, on most modern operating systems. This dataset covers an additional major version of the NetBackup software compared to the dataset in Section 2.2, so it also covers twice as many installations.
Chapter 2. Storage maintenance in the field

<table>
<thead>
<tr>
<th>Report type</th>
<th>Metrics used in study</th>
</tr>
</thead>
<tbody>
<tr>
<td>Installation</td>
<td>Installation time</td>
</tr>
<tr>
<td>Runtime report</td>
<td>Job information: starting time, type, size, number of files, client policy, deduplication ratio, retention period</td>
</tr>
<tr>
<td>Domain report</td>
<td>Number and type of policies, number of clients, number of storage media, number of storage servers and appliances</td>
</tr>
</tbody>
</table>

Table 2.8. Telemetry report metrics used in the study.

<table>
<thead>
<tr>
<th>Job type</th>
<th>Percentage of jobs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Incremental Backups</td>
<td>45.27%</td>
</tr>
<tr>
<td>Full Backups</td>
<td>31.20%</td>
</tr>
<tr>
<td>Snapshot Operations</td>
<td>12.61%</td>
</tr>
<tr>
<td>Management Operations</td>
<td>10.12%</td>
</tr>
<tr>
<td>Recovery Operations</td>
<td>0.80%</td>
</tr>
</tbody>
</table>

Table 2.9. Breakdown of all jobs in the dataset by type.

**Monitoring duration.** The backup domains included in our study were each monitored for 5.5 months on average, and up to 32 months. As in Section 2.2, the monitoring time is not always equivalent to the total lifetime of the domain, and many of these domains were still online at the time we conducted our analysis.

**Architecture.** While NetBackup supports the 3-tiered architecture model, only 35% of domains in our dataset use dedicated storage servers. The remaining domains omit that layer, opting for a 2-tier system instead. Additionally, while backup software can be installed on any server, storage companies also offer Purpose-Built Backup Appliances (PBBAs) [107]. 31% of domains in our dataset represent this market by deploying NetBackup on Symantec PBBAs.

### 2.3.4 Job scheduling

While the master server can reorder policy jobs to increase overall system efficiency, it adheres to user preferences that dictate when, and how often a job should be scheduled. This section looks into the way that these parameters are configured by users across backup domains, and the workload generated in the domain as a result.

**Job types**

Recall from Section 2.3.2 that policies consist of a predefined series of operations, each carried out by a separate job. We collected data from 209.5 million jobs, and we group them in five distinct categories: full and incremental backups, snapshots, recovery, and management operations. In Table 2.9, we show a breakdown of all jobs in our dataset by job type. Across all monitored backup domains, we find that 76% of jobs perform data backups, having processed a total of 1.64 Exabytes of data, while 13% of jobs


Figure 2.15. Distribution of the average scheduling frequency of different job types across backup domains. Recovery operations are broken into two groups of domains with more, and less than 5 recovery operations each. Despite being of similar size, the characteristics of each group differ significantly.

A factor indicative of data churn in a backup domain is the rate at which jobs are scheduled to backup, restore, or manage backed-up data. To quantify the scheduling frequency of different job types for a given domain, we rely on the starting times of individual jobs. Specifically, starting times are used to estimate the average occurrence rate of different jobs of each domain policy, on individual clients. In Figure 2.15, we show the scheduling frequency distributions of different job types across backup domains.

Overall, we find that the average frequency of recovery operations differs depending on their number. In Figure 2.15, we show the distributions of the recovery frequency for two domain groups having recovered data more, and less than 5 times. The former group consists of 337 domains that recovered data 17 times on average, and the latter consists of 262 domains with 3 recovery operations on average. By definition, our analysis excludes an additional 676 domains that initiate recovery only once. For domains with multiple events, the distribution of their frequency spans 1-2 weeks, with an average of 6 days. On the other hand, domains with fewer recovery operations perform them significantly less frequently, up to 2 months apart and every 24 days on average. Since recovery operations are initiated manually by users, we have no accurate way of pinpointing their cause. These results, however, suggest that frequent recovery operations may be attributed to disaster recovery testing, while infrequent ones may be due to actual disasters. Interestingly, both domain groups are equally small, but when domains with a single recovery event are factored in, the group of infrequent recovery operations doubles in size.
Figure 2.16. Tukey boxplots (without outliers) that represent the average size of full backup jobs, for different job scheduling frequencies. Means for each boxplot are also shown. Frequent full backups seem to be associated with larger job sizes, suggesting that they may be preferred as a response to high data churn.

In the case of backup jobs, the general belief is that systems in the field rely on weekly full backups, complemented by daily incremental backups [47, 120, 259]. Our results confirm this assumption for incremental backups, which take place every 1-2 days in 81% of domains. Daily incremental backups are also the default option in NetBackup. For full backups, however, our analysis shows that only 17% of domains perform them every 6-8 days on average. Instead, the majority of domains perform full backups more often: 15% perform them every 1-2 days, and 57% perform them every 2-6 days. This is despite the fact that weekly full backups is the default option. As expected, management operations take place on a daily or weekly basis, since they usually follow (or precede) an incremental or full backup operation. Snapshot operations display a similar trend to full backups, as they are mostly used by clients in lieu of the latter.

Of the 65% of domain policies that perform full backups every 6 days or fewer, only 33% also perform incremental backups at all. On the other hand, 76% of policies that perform weekly full backups also rely on incremental backups. To determine whether full backups are performed frequently to accommodate high data churn, we group average full backup sizes per client policy according to their scheduling frequency, and present the results as a series of boxplots in Figure 2.16. Note that regardless of frequency, full backups tend to be small (medians in the order of a few gigabytes), due to the efficiency of deduplication. However, the larger percentiles of each distribution show that larger backup sizes tend to occur when full backups are taken more frequently than once per week. While this confirms our assumption of high data churn for a fraction of the clients, the remaining small backup sizes could also be attributed to overly conservative configurations, a sign that policy auto-configuration is an important feature for future data protection systems.

Scheduling windows

Observation 13: Users prefer default scheduling windows during weekdays, resulting in nightly bursts of activity. Default values are overridden, however, to avoid scheduling jobs during the weekend.
Another important factor for characterizing the workload of a backup system is the exact time jobs are scheduled. A popular belief is that backup operations take place late at night or during weekends, when client systems are expected to be idle [55, 243]. In Figure 2.17, we show our findings for all the jobs in our dataset. The presented density function was computed by normalizing the number of jobs that take place in a given domain, to prevent domains with more jobs from affecting the overall trend disproportionately. We note that this normalization had minimal effect on the result, which suggests that the presented trend is common across domains.

The hourly scheduling frequency is similar for each day, although there is less activity during the weekend. We also find that the probability of a job being scheduled is highest starting at 6pm and 12am on a weekday. We attribute the timing of job scheduling to customers using the default scheduling windows suggested by NetBackup, which start at 6pm and 12am every day. The choice to exclude weekends, however, seems to be an explicit choice of the user. This result suggests that automated job scheduling, where the only constraints would be to leverage device idleness [26, 84, 151, 152], would be more practical, allowing the system to schedule jobs so that such activity bursts are avoided.

While Figure 2.17 merges all job types, different jobs exhibit different scheduling patterns, as shown in Figure 2.15. Our data, however, does not allow a matching of job types to scheduling times at a granularity finer than the day on which the job was scheduled. Thus, we partition jobs based on their type, and in Figure 2.18 we show the probability that a job of a given type will be scheduled on a given day of the week. We find that incremental backups are scheduled to complement full backups, as they tend to get scheduled from Monday to Thursday, while full backups are mostly scheduled on Fridays. Note that the latter does not contradict our previous result of full backups that take place more often than once a week, as the probability of scheduling full backups any other day is still comparatively high. Recovery operations also take place within the week, with a slightly higher probability on Tuesdays (which we confirmed as not related to Patch Tuesday [154]). Finally management operations do not follow any particular trend and are equally likely to be scheduled on any day of the week.
Chapter 2. Storage maintenance in the field

2.3.5 Backup data growth

Characterizing backup data growth is crucial for estimating the amount of data that needs to be transferred and stored, which allows for efficient provisioning of storage capacity and bandwidth. Towards this goal, we use the periodic telemetry reports to quantify the growth rate of the number of clients across domains. We also analyze the sizes and number of files of different job types, and their deduplication ratios across backup domains. Finally, we look into the time that backup data is retained.

Client growth rate

Observation 14: The number of clients in a domain increases by an average of 7 clients every 3.7 months.

Clients are the producers of backup data, and the consumers of said data during recovery. As a result, the number of jobs running on a backup domain is directly proportional to the number of clients in the domain, making it important to quantify the rate at which their population grows over time.

Once the initial configuration period for a backup domain has elapsed, we find that clients tend to be added to, or removed from the domain in groups. Therefore, we characterize a domain’s client population growth by quantifying the average rate of change in the client population, the sign indicating an increase or decrease in the population, and size of each change.

To estimate the rate at which the number of clients change, we extract inter-arrival times between changes through change-point analysis [123], a cost-benefit approach for detecting changes in time series. Then, we estimate the average rate of change for a domain as the average of these inter-arrival times. In Figure 2.19, we show the distribution of the average rates of change, i.e. the average number of months between changes in the number of clients across domains. For 42% of backup domains, the number of clients remains fixed after the first 3 weeks of operation, while on average the number of clients in a domain changes every 3.7 months. Overall, we find no strong correlation between the rate of change in the number of clients, and the domain’s lifetime.

We further analyze the sign and size of each population change. Of all events in which a domain’s
client population changes, 93% are attributed to the addition of clients. However, 78% of domains never remove clients. Regarding the size of each change, Figure 2.20 shows the distribution of the average number of clients involved in each change, across all domains in our study. On average, a domain’s population changes by 7.3 clients at a time. The average standard deviation of the number of clients over time is 13.1% of the corresponding expected value, indicating low variation overall. However, the 95% confidence intervals (C.I.) for each mean in Figure 2.20, suggest that growth spurts as large as 2.16 times the average value are possible, as this is the width of the average 95% confidence interval.

Job sizes and number of files

Observation 15: Incremental and full backups tend to be similar in size and files transferred, due to the effectiveness of deduplication, or misconfigurations. Recovery jobs restore either a few files, or entire volumes.
Figure 2.21. Distribution of the average job size of a given job type across backup domains, after the data has been deduplicated at the client side. Incremental backups resemble full backups in size. Snapshot operations are not included, as they do not incur data transfer.

An obvious factor when estimating a domain’s data growth is the size of backup jobs. To mitigate this growth, modern backup systems use deduplication and incremental backups. In most systems, deduplication is implemented using a client-server model [91]. The client component is responsible for initiating the backup process, performing I/O, calculating file segment boundaries and their fingerprints, and initiating fingerprint lookups at the server. Data transfers are initiated only for segments with new fingerprints. The server is responsible for backup and file metadata indexing and storage, as well as organizing the fingerprint index. In the case of incremental backups, the backup system needs to determine which files have changed since the last backup took place. This is achieved by using the modification and creation times of each file, adjusted by the clock skew between the client and the server [220, p. 343].

In Figure 2.21, we show the distributions of the average number of bytes transferred for different job types across all domains, after the data has been deduplicated at the client. Averages for each operation are shown in the legend, and marked on the x axis. Snapshot operations are not included, as they do not incur data transfer.

Surprisingly, incremental backups resemble full backups in size. Although the distribution of full backups is skewed toward larger job sizes, 29% of full backups on domains that also perform incremental backups tend to be equal or smaller in size than the latter, 21% range from $1 - 1.5$ times the size of incremental backups, and the remainder range from $1.5 - 10^6$ times. We attribute the small size difference to three reasons. First, systems with low data churn can achieve high deduplication rates, which are common as we show later in this section. Second, misconfigured policies or volumes that do not support incremental backups often fall back to full backups, as suggested by support tickets. Third, maintenance applications, such as anti-virus scanners, can update file metadata making unchanged files appear modified. Overall, the average backup job sizes in Figure 2.21 are 5-8 times smaller than the file sizes reported by Wallace et al. [243], likely due to their study considering the sizes of all files in the file system storing the backup images.

Since recovery operations can be triggered by users to recover an entire volume or individual files, the distribution of recovery job sizes is not surprising. 32% of recovery jobs restore less than 1GB, while the
average job can be as large as 51GB. Finally, management operations, which consist mostly of metadata backups (95.7%), but also backup image (1.5%) and snapshot (2.8%) duplication operations, are much smaller than all other operations, as expected.

Figure 2.22 shows the distributions of the average number of files transferred for different job types in each domain. Similar to job sizes, the average number of files transferred per incremental backup is 31% smaller than that for full backups, and both job types are characterized by similar CDF curves. Recovery operations transfer as many files as full backups on average, yet the majority transfer fewer than 200 files. This is in line with our results on recovery job sizes. Given that large recovery jobs also occur less frequently, these results suggest that most recovery operations are not triggered as a disaster response, but rather to recover data lost due to errors, or to test the recoverability of backup images. Management operations, being mostly metadata backups, transfer significantly fewer files than other job types on average.

Deduplication ratios

Observation 16: Deduplication can result in the reduction of backup image sizes by more than 88%, despite average job sizes ranging in the tens of gigabytes.

For clients that use NetBackup’s deduplication solution, we analyzed the daily deduplication ratios of jobs, i.e. the percentage by which the number of bytes transferred was reduced due to deduplication. Figure 2.23 shows the distributions of the average daily deduplication ratio for management operations, full, and incremental backups across backup domains. Recovery and snapshot jobs are not included as the notion of deduplication does not apply. Since deduplication happens globally across backup images, deduplication ratios for backups tend to increase after the first few iterations of a policy. In general, sustained deduplication ratios as high as 99% are not unusual. Across all domains in our dataset, however, the average daily deduplication ratio is 88-89%, for both full and incremental backups. It is interesting to note that despite such high deduplication ratios, jobs in the order of tens of gigabytes are common (as shown in Figure 2.21), suggesting that even for daily incremental jobs, the actual job sizes are an order of magnitude larger in size. These results are in agreement with previous work [243], which
Figure 2.23. Distributions of the average daily deduplication ratio of different job types, across backup domains. Incremental and full backups observe high deduplication ratios, while the uniqueness of metadata backups (management operations) makes them harder to deduplicate.

reports average deduplication ratios of 91%.

Finally, for management operations the average deduplication ratio is 68%. Since only 1.1% of domains that use deduplication enable it for management operations, we do not attach much importance to this result. For the reported domains, however, it can be attributed to the uniqueness of metadata backups, which do not share files with other backup images on the same backup domain and consist of large binary files.
Chapter 3

Scheduling maintenance I/O

As described in earlier chapters, maintenance work today is reserved for downtime periods during which the system is suspended from normal operation. This way, administrators can easily avoid any impact of maintenance on user applications. Unfortunately, downtime is expensive, as the system needs to remain offline. As a result, it is costly to overprovision. At the same time, estimating the amount of time to complete all maintenance work can be challenging, as it depends on the data growth rate in the system, hardware characteristics, and the maintenance tasks involved. In addition to provisioning, maintenance downtime can be non-trivial to schedule conveniently. So far, administrators seem to prefer late nights and weekends (see Section 2.3.4), to avoid interference with users. This assumption is becoming less applicable in shared environments, such as the cloud, where multiple time zones are at play.

Another approach that is gaining more traction recently is to schedule maintenance work during times when a storage device is idle [26, 66, 84, 151, 152]. In order to achieve this, device idleness must be accurately predicted. To understand idleness, we performed a statistical analysis of publicly available traces of 77 disk workloads in enterprise environments [119, 216], representing a diverse set of workload types. While our analysis is not the first of its kind [151, 185], it is the most extensive both in terms of the types of workloads represented, and their duration. We have applied the observations of our analysis towards the design of idle time predictors, and using the most accurate predictors, we describe an approach that maximizes maintenance throughput for a given workload, while meeting a predefined slowdown goal for I/O requests issued by user applications.

To assess the applicability of our scheduling approach, we apply it in the context of scrubbing, a maintenance task widely used to provide guarantees on the reliability of storage devices [29, 72, 127, 163, 164]. Specifically, most commercial storage systems use a “scrubber” to protect against LSEs (see Section 2.1.2): a background process that periodically performs full-disk scans to proactively detect and correct LSEs. The goal of a scrubber is to minimize the time between the occurrence of an LSE and its detection/correction, also referred to as the Mean Latent Error Time (MLET), since during this time the system is exposed to the risk of data loss (e.g. if another drive in the disk array fails). In addition to reducing the MLET, a scrubber must ensure that it does not significantly affect the performance of foreground workloads running on the system.

The importance of employing an efficient scrubber will only increase in the future, as growing disk capacities increase both the overhead of a full disk scan, and the rate at which LSEs happen. Unfortunately, the scrubbers employed in today’s storage systems are quite simplistic: they scan the disk
sequentially in increasing order of Logical Block Numbers (LBN) at a rate determined by the system administrator. This simple approach ignores a number of design options that have the potential for reducing the MLET, as well as the impact on foreground workload.

The first design question is determining the order in which to scrub the disk’s sectors. While scrubbing the disk sequentially is simple and efficient (as sequential I/O is more efficient than random accesses), recent research [163] shows that an alternative approach, called staggered scrubbing, provides lower MLET. Staggered scrubbing aims to exploit the fact that LSEs happen in (temporal and spatial) bursts: rather than sequentially reading the disk from beginning to end, the idea is to quickly “probe” different regions of the drive, hoping that if a region has a burst of errors the scrubber will detect it quickly and then immediately scrub the entire region. While reducing MLET, the overhead of the random I/O in staggered scrubbing can potentially reduce scrub throughput and increase the impact on foreground workloads. Unfortunately, there exists no experimental evaluation that quantifies this overhead, and staggered scrubbing is currently not used in practice. A number of other implementation choices for a scrubber have not been well studied, including the choice of a user-level versus kernel-level implementation, or the effect of the disk interface (SCSI versus ATA). To explore this space, we developed a framework that can be used to implement scrubbing algorithms in only tens of lines of code within the Linux kernel, and made its source code publicly available. Using our framework, we provide the first implementation of a staggered scrubber and compare its performance to that of a standard sequential scrubber. We also implemented a user-level scrubber to allow for a quantitative comparison between user-level and kernel level scrubbing.

The second design question is deciding when to issue scrub requests. The scrubbers employed in commercial storage systems today are either scheduled during periods when the system has been taken offline for maintenance, or issue requests at a predefined rate, e.g., every $r$ msec. The former approach is costly and, as we showed in Chapter 2, it is not trivial for administrators to provision sufficient downtime. The latter approach has two shortcomings. First, it does not attempt to minimize impact on foreground traffic, as scrub requests are issued independently of foreground activity. Second, it is often hard for system administrators to choose the rate $r$ that is right for their system, since it is hard to predict the impact on foreground traffic that is associated with a particular rate. Similarly, a parameter that is not very well understood is the scrub request size, or the number of sectors scrubbed by individual scrub requests. Larger request sizes lead to more efficient use of the disk, but also have the potential of bigger impact on foreground traffic, as foreground requests that arrive while a scrub request is in progress get delayed. Using the observations and predictors derived from our analysis of disk idleness, we show that our scheduling approach outperforms the default Linux I/O scheduler, the only one to allow for I/O prioritization, and we provide conclusive answers to the questions of when should maintenance I/O requests be issued, and at what size, in order to achieve the highest throughput for a given slowdown of the foreground workload.

The rest of the chapter is organized as follows. In Section 3.1 we present our statistical analysis of idleness, as well as our idleness predictors and their evaluation. Section 3.2 describes the framework we implemented and used to develop sequential and staggered scrubbing; a performance evaluation of the two algorithms follows in Section 3.3, where we also demonstrate the efficiency of our idleness predictors.
3.1 Understanding idleness

To conclusively answer the questions of when should maintenance requests be issued and at what size, we perform an extensive statistical analysis of publicly available traces (Section 3.1.1). Through this analysis, we identify statistical properties of I/O workloads that aid us in deriving better techniques for scheduling maintenance requests. We define and experimentally evaluate those techniques in Sections 3.1.2 and 3.1.3.

3.1.1 Insights from the statistical analysis of traces

While there exists previous work that has provided some general analysis of I/O workloads [185, 187], in this section we examine an extensive collection of workload types that are considerably longer in duration. Furthermore, we look at I/O traffic characteristics that are relevant to scheduling maintenance requests in the background. The main goal is to schedule maintenance requests with minimal impact on foreground traffic by making better use of the idle intervals that hard drives experience.

We have worked with a set of 77 disk traces in total, spanning from one week to one year (we make use of only one week in our experiments) and being used in almost all possible scenarios: home and project directories, web and print servers, proxies, backups, etc. These traces are available publicly from SNIA [216]. Although we ran the experiments in the rest of the chapter for almost all disks, we have chosen to focus mainly on a few disks from each trace collection when presenting our results. These disks contain the largest number of requests per week, and represent diverse workloads. The characteristics of the chosen traces are summarized in Table 3.1.

<table>
<thead>
<tr>
<th>Trace</th>
<th>Disk</th>
<th>Requests</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSR Cambridge (2008)</td>
<td>src11</td>
<td>45,746,222</td>
<td>Source Control</td>
</tr>
<tr>
<td></td>
<td>usr1</td>
<td>45,283,980</td>
<td>Home dirs</td>
</tr>
<tr>
<td></td>
<td>proj2</td>
<td>29,266,482</td>
<td>Project dirs</td>
</tr>
<tr>
<td></td>
<td>prn1</td>
<td>11,233,411</td>
<td>Print server</td>
</tr>
<tr>
<td>HP Cello (1999)</td>
<td>c6t8d0</td>
<td>9,529,855</td>
<td>News Disk</td>
</tr>
<tr>
<td></td>
<td>c6t5d1</td>
<td>4,588,778</td>
<td>Project files</td>
</tr>
<tr>
<td></td>
<td>c6t5d0</td>
<td>3,365,078</td>
<td>Home dirs</td>
</tr>
<tr>
<td></td>
<td>c3t3d0</td>
<td>2,742,326</td>
<td>Root &amp; Swap</td>
</tr>
<tr>
<td>MS TPC-C (2009)</td>
<td>disk66</td>
<td>513,038</td>
<td>TPC-C run</td>
</tr>
<tr>
<td></td>
<td>disk88</td>
<td>513,844</td>
<td>TPC-C run</td>
</tr>
</tbody>
</table>

Table 3.1. SNIA Block I/O traces used in the paper

Periodicity. Periodic behavior in the length of idle intervals is helpful for scheduling maintenance requests, since it provides a predictable pattern that a scheduler can exploit. While we are not aware of prior work that has specifically focused on periodic behavior of I/O workloads, one might expect I/O workloads to exhibit periodic patterns, such as diurnal trends. We begin our analysis of periodicity by a visual inspection of how the request arrival rate varies as a function of time. Figure 3.1 plots the number of requests per hour as a function of time for four representative traces from our trace collection. We observe that all four traces exhibit repeating patterns, often with spikes at 24 hour intervals, but in some
cases also at other time intervals. For the *HP Cello* traces, these consistent spikes could be attributed to daily backups \[195\], while for the *MSR Cambridge* traces, activity peaks on different hours for different disks, with some days seeing smaller or no spikes. We believe that such activity spikes can be common in the field, with varying intensity based on their causes that could be anything from scheduled tasks to applications, and/or human activity.

For a more statistically rigorous approach to detecting periodicity, we used analysis of variance (ANOVA) to identify the time interval with the strongest periodic behavior for each trace in our data set. The results are shown in Figure 3.2. We observe that for most traces ANOVA does identify periods, most commonly at intervals of 24 hours (our analysis was done at the granularity of hours, so periods of one hour in Figure 3.2 mean that no periodicity was identified).

**Autocorrelation.** Autocorrelation is an interesting statistical property, as it means that the length of previous (recent) idle intervals is predictive of the lengths of future idle intervals. This is the exact kind of information that a scheduler could use to identify long idle intervals, in which to schedule maintenance requests. Previous work \[185\] has reported evidence of autocorrelation for some (not publicly available)
Table 3.2. SNIA Trace idle interval duration analysis results

<table>
<thead>
<tr>
<th>Trace</th>
<th>Disk</th>
<th>Mean (s)</th>
<th>Variance</th>
<th>CoV</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSR Cambridge</td>
<td>src11</td>
<td>0.4640</td>
<td>101.31</td>
<td>21.693</td>
</tr>
<tr>
<td>(2008)</td>
<td>usr1</td>
<td>0.0997</td>
<td>0.7448</td>
<td>8.6516</td>
</tr>
<tr>
<td></td>
<td>proj2</td>
<td>0.1384</td>
<td>772.18</td>
<td>200.75</td>
</tr>
<tr>
<td></td>
<td>prn1</td>
<td>0.2280</td>
<td>8.3073</td>
<td>12.641</td>
</tr>
<tr>
<td>HP Cello</td>
<td>c6t8d0</td>
<td>0.1502</td>
<td>4.3243</td>
<td>13.845</td>
</tr>
<tr>
<td>(1999)</td>
<td>c6t5d1</td>
<td>0.4503</td>
<td>180.13</td>
<td>29.807</td>
</tr>
<tr>
<td></td>
<td>c6t5d0</td>
<td>0.4345</td>
<td>15.545</td>
<td>9.0731</td>
</tr>
<tr>
<td></td>
<td>c3t3d0</td>
<td>0.4555</td>
<td>14.051</td>
<td>8.2301</td>
</tr>
<tr>
<td>MS TPC-C</td>
<td>disk66</td>
<td>0.0014</td>
<td>1.5e-6</td>
<td>0.8608</td>
</tr>
<tr>
<td>(2009)</td>
<td>disk88</td>
<td>0.0015</td>
<td>1.6e-6</td>
<td>0.8785</td>
</tr>
</tbody>
</table>

Figure 3.3. What fraction of a disk’s idle time do the largest idle periods make up? Note that the $x$ axis is cut off at the 50th percentile.

disk traces in the form of Hurst parameter values larger than 0.5. We studied the autocorrelation function for all our traces and found that 44 out of the busiest 63 disk traces exhibit strong autocorrelation.

**Decreasing hazard rates and long tails.** Previous work [185] has reported that the distribution of I/O request inter-arrival times exhibits high variability. We verified that this is also the case for our traces. We observe Coefficients of Variation\(^1\) typically in the 10-30 range (Table 3.2), and in one case as high as 200. These numbers are even higher than those reported by Riska and Riedel [185], who observed a CoV of 19 for their most variable trace. To put those numbers in perspective, recall that an exponential distribution has a CoV of 1. The exponential distribution is a memoryless distribution, that is, if idle times were exponentially distributed, then the expected time until the next request arrival would always be the same, independent of the time since the last arrival. We believe this is unrepresentative of real workloads, and found it to be the case only for the TPC-C traces [216, 229].

The high CoV values signify long tails in the distribution of idle intervals. In the context of our work, a long tail implies that a large fraction of the system’s total idle time is concentrated in a small

\(^1\)Recall that the Coefficient of Variation (CoV) is defined as the standard deviation divided by the mean.
fraction of very long idle intervals. For a closer study of this characteristic, Figure 3.3 plots the fraction of the total idle time in the system that is made up by the longest idle intervals, that is, each data point \((x, y)\) shows that the \(x\)% largest idle intervals in the system account for \(y\)% of the total idle time in the system. We observed that for all traces a very large fraction of idle time is concentrated in the tail of the idle time distribution: typically more than 80% of the idle time is included in less than 15% of the idle intervals, in most cases the skew is even stronger.

The strong weight in the tail of the distributions is good news for scheduling background workloads. It means that if we can identify the 15% largest intervals, we can make use of more than 80% of the total idle time in the system. Scheduling background work for only a small percentage of the idle intervals is advantageous for limiting the number of possible collisions, where a foreground request arrives while a background request is in progress.

An obvious problem that occurs at this point is identifying at the beginning of an idle interval, whether it is going to be one of the few very large ones (and hence, maintenance work should be initiated). We can derive two ideas for identifying long idle intervals from our previous two observations: since there is periodicity in the data, we could schedule background work only during those times of the day that tend to be lightly loaded; alternatively, since there is strong autocorrelation in the data, we could use auto-regression to predict the length of upcoming idle intervals based on the lengths of previous ones.

The high CoVs we observed in our traces suggest a third option for predicting idle times: it is possible that the CoVs in our traces are so much higher than that for an exponential distribution, because their empirical idle time distributions have decreasing hazard rates, i.e., the longer the system has been idle, the longer it is expected to stay idle. In this case, an approach based on waiting might work well, where we wait until the system has been idle for a certain amount of time before we issue a maintenance request.

To check for decreasing hazard rates we plotted the expected remaining idle time, as a function of how long the disk has been idle in Figure 3.4. A data point \((x, y)\) in the graph means that after being idle for \(x\) seconds, the system will be idle in expectation for an additional \(y\) seconds before the next request arrives. We observe that the lines for all Cello and MSR traces are continuously increasing with the exception of the TPC-C traces, again showing that they are unrepresentative of real workloads\(^2\). In

\footnote{This is to be expected, as a benchmark is designed to test performance, driving the system with continuous high load}
fact, having been idle for a long time increases the expected remaining idle time by several orders of magnitude (note the log scale on the y-axis). Since the expected remaining idle time is just an average (on average after waiting for $x$ seconds it will take another $y$ seconds until the next foreground request arrives) which might be biased by outliers, we also plotted the first percentile of remaining time in Figure 3.5. A data point $(x, y)$ in this graph means that in 99% of the cases, after waiting for $x$ seconds we have at least another $y$ seconds before the next foreground request arrives. We again note strongly increasing trends.

One potential problem with the wait-based approach is that we miss out on using the idle time that passes while we wait. Figure 3.6 plots the fraction of the total idle time in the system that we can exploit if we only schedule maintenance requests after waiting for $x$ seconds. The figure shows that even after waiting for some time before issuing maintenance requests, we can still make use of a significant fraction of the total idle time. For example, for a wait time on the order of 100 ms we can still make use of more than 60–90% of the total idle time in the system, depending on the workload. At the same time, the number of possible collisions between arriving foreground and background requests is limited, since less than 10% of all idle intervals in our traces are larger than 100 ms and will be picked to perform maintenance work.

Finally, our observation of decreasing hazard rates has another implication for designing a scheduler
for maintenance requests. Existing approaches for scheduling background requests [84, 151] consist of a method for identifying a starting criterion (when to start issuing background requests), and a stopping criterion (when to stop issuing background requests). Figure 3.4 tells us that we need not worry about a stopping criterion. The goal of background scheduling policies is to schedule background requests when the chance of a foreground request arriving is low. Decreasing hazard rates, however, imply that the chance of a foreground request arriving at any given moment diminishes with time, past the beginning of the idle interval. Therefore, once maintenance work is initiated, if the system is still idle upon the completion of a maintenance request, the chance of a foreground request arriving is even lower than before issuing the maintenance request. This means that once an idle interval is identified as long, the policy that makes most sense statistically is to keep sending background requests, until the next foreground request arrives. Finally, we find that the distribution of request inter-arrival times in the TPC-C benchmark traces deviates significantly from the distributions in real workload traces. As a result, we exclude these traces from the analyses in the remainder of this chapter.

3.1.2 Profiting from idleness

In this section we define and evaluate three different policy classes for scheduling maintenance requests, which have all been directly motivated by our results in Section 3.1.1.

**Autoregression (AR) – Learning from the past**

The strong periods and autocorrelation we observed in our analysis in Section 3.1.1 motivated us to look into approaches that capture repetitive patterns. We examined autoregressive (AR) models, which use successive observations of an event to express relationships between a dependent and one or more independent variables. In our experiments we used the simple AR($p$) model, which regresses a request inter-arrival interval of length $X_t$ against past intervals $X_{t-1}, ..., X_{t-p}$:

$$
X_t = \mu + \sum_{i=1}^{p} a_i(X_{t-i} - \mu) + \epsilon_t,
$$

where $a_1, ..., a_p$ are parameters of the model, $\epsilon_t$ is white noise, and $\mu$ expresses a mean calculated over past intervals. We estimate the order $p$ using Akaike’s Information Criterion [7], that optimizes the ratio between the number of parameters and the accuracy of the resulting model. Since AR models can only be applied to regular time series, i.e., sequences of events recorded at regular intervals, we model the durations of request inter-arrival intervals [93]. This also implies that AR predictions are estimations of the amount of time until the arrival of the next request. We attempted to fit several AR models to our data, including ACD [74] and ARIMA [146], and found that AR($p$) is the only model that can be fitted quickly and efficiently to the millions of samples that need to be factored at the I/O level.

Our AR policy works by predicting the length of the current idle interval $X_t$ based on previous intervals $X_{t-1}, ..., X_{t-p}$ using the AR($p$) model. The policy makes the prediction for $X_t$ at the beginning of the current idle interval and starts firing maintenance requests if the prediction $X_t$ is larger than some specified time threshold $c$, which is a parameter of the policy. Once it starts issuing maintenance requests it continues until a foreground request arrives.
Figure 3.7. Comparison of the Auto-regression and Waiting approaches with the optimal (Oracle).

Waiting – Playing the waiting game

The decreasing hazard rates in our traces imply that after the system has been idle for a while, it will likely remain idle. This property is exploited by the Waiting policy, which dictates that no requests are to be issued, unless the system has remained idle for some time $t$ ($t$ is a parameter of the policy). Requests stop being issued only upon the arrival of a foreground request.

AR+Waiting – Combining auto-regression and waiting

This policy combines the Auto-regression and Waiting approaches. It waits for a time threshold $t$ and if the system has been idle for that long it starts firing, provided that the auto-regression prediction for the length of this idle interval is larger than some time threshold $c$.

Comparison of policies

Naturally, for all policies there is a trade-off in the throughput that the maintenance task achieves, and the resulting impact on the performance of the foreground traffic. The trade-off is governed by the parameters of the policies, so choosing larger values for parameters $c$ and $t$ of the AR and Waiting policies,
respectively, will lead to lower impact on the foreground traffic at the cost of reduced maintenance throughput. For a fair comparison of the different policies, we need to find which one achieves highest maintenance throughput for a given fixed penalty to the foreground traffic.

The reason we try to optimize for maintenance throughput, is because we attempt to make no assumptions about the amount of maintenance that needs to be performed. This is because scrubbing, unlike other maintenance tasks such as backup and defragmentation, accesses only cold data which is often orders of magnitude more than the data accessed by the workload. As a result, it cannot be estimated by using workload I/O traces. In this case, the maximum throughput metric provides an estimate of the amount of data that can be scrubbed in a specific period, e.g. weekly. Administrators can then use this metric to gauge whether this is acceptable for their system.

We compare policies by varying their corresponding parameters, and plotting the resulting amount of idle time that can be utilized for maintenance versus the number of resulting collisions (the fraction of foreground requests that are delayed due to a maintenance request in progress). The results are shown in Figure 3.7: MSRusr2 in Figure 3.7a is representative of most disks in our trace collections, while HPc6t8d0 in Figure 3.7b is characterized by multiple short idle intervals, representing a worst case scenario with respect to collisions. The numbers on top of the data points provide the parameter setting used to derive that data point: the wait time threshold $t$ for the Waiting approach and the threshold $c$ for the AR approach.

In addition to AR and Waiting, we also plot results for the combined approach. For that, we experiment with four different values of the $c$ parameter for the AR component. These four values were chosen to be the $20^{th}$, $40^{th}$, $60^{th}$, and $80^{th}$ percentile of all observed AR values for a trace. For each of these values we plot one line in the graph, which results from varying the wait time threshold.

Finally, for reference we also plot the best possible results that could be achieved by a clairvoyant Oracle that can accurately predict the $x\%$ longest idle intervals and only utilize those, maximizing the amount of idle time one can utilize for a collision rate of $x\%$. This Oracle provides an upper bound on the results one can hope to achieve.

The results in Figure 3.7 are quite interesting. We find that the simple Waiting approach clearly outperforms AR and the combined approaches, as it consistently manages to utilize more idle time for a given collision rate. On the other hand, the pure AR policy shows by far the worst performance, which we attribute to its inability to capture enough request history to make successful decisions. While outperforming the other policies, the Waiting approach is weaker than the Oracle. One might wonder whether this is due to the fact that Waiting fails at predicting all the long idle intervals, or because it wastes idle time while sitting idle waiting for $t$ time before firing. To answer this question we also plotted results for a hypothetical policy, Lossless Waiting, which utilizes the same intervals as Waiting, while assuming that we can magically also make use of the time that passes while waiting. We see that this hypothetical policy performs very closely to the best possible policy (the Oracle). This means that the Waiting approach is extremely good at identifying long idle intervals, and only falls short from achieving optimal possible performance due to the time spent waiting.

The real impact of metrics such as the collision rate of maintenance with foreground I/O depends on the characteristics of both the workload and the hardware. This is because the delay of foreground I/O in the case of collision will be affected by both the proximity of foreground to maintenance I/O, and the effect this proximity has on device performance (e.g., time amplification due to additional seek time). In Section 3.3 we explore the practical impact of this metric by replaying real workload traces.
3.1.3 Sizing up throughput opportunities

One important remaining parameter is the size of each maintenance request. While larger maintenance requests will increase the throughput of the maintenance task, they also increase the impact on foreground workloads, as collisions become more expensive (increasing delays for the foreground request arriving while a maintenance request is in progress, and often also for the ones following that). Our goal is to take as input from a system administrator the average and maximum tolerable slowdown per foreground application request, and within these limits find the parameters that maximize maintenance throughput.

Figure 3.8 shows that the throughput a maintenance task can achieve while limiting the slowdown of the foreground application to some acceptable threshold, can vary dramatically as a function of the request size. The fixed lines in Figure 3.8 were obtained by simulating the Waiting policy while keeping the request size fixed at 64 kb, 768 kb, 1216 kb, 1280 kb and 4 MB, and varying the wait time threshold. For each threshold value, we plot the resulting average slowdown on foreground requests versus the throughput achieved by the maintenance task (we experimented with request sizes ranging from 64 kb to 4 MB and a maximum allowed request slowdown of 50 ms, but plot only results for 5 sizes for readability).

We observe that for the two extreme examples of 64 kb and 4 MB, the maintenance task using 4 MB requests achieves a consistently higher throughput for the same foreground application slowdown. This motivated us to try and determine the optimal maintenance request size for a specific request slowdown, i.e. the request size that will lead to a slowdown within the prescribed limit, while maximizing the throughput of the maintenance task (we focus only on the Waiting approach, since we showed how it outperforms the rest). We accomplished this using simulation to find the optimal request size in the range from 64 KB to 4 MB (bounded by the maximum tolerable slowdown), and the wait time threshold for which it yields the maximum throughput, satisfying the given average slowdown goal per I/O request. For each size, the optimal threshold can be found efficiently through binary search, since for a given request size, larger thresholds will always lead to smaller slowdowns. Using the threshold we can then estimate the maximum throughput per request size, and use that as a comparison metric to find the optimal request size. To summarize, our policy chooses for each slowdown the request size and the wait time threshold that maximize the maintenance task’s throughput. Results for our policy are shown in
Figure 3.8, where we find, for example, that some optimal \((\text{slowdown}, \text{request size})\) pairs are: \((0.5\text{ms}, 768\text{KB})\), \((1.0\text{ms}, 1280\text{KB})\) and \((1.5\text{ms}, 1216\text{KB})\). We observe that this approach performs significantly better than either the 64KB or the 4MB approach.

In the experiment above, we limit ourselves to picking the best request size and wait time threshold to maximize throughput for a given acceptable slowdown, and have the maintenance task send back-to-back requests of that fixed size until a collision occurs. Based on our observations of decreasing hazard rates in Section 3.1.1, one might think of doing even better by varying the request size over time, as the maintenance task fires. When the wait time has just ended and the task starts sending requests, the probability of a foreground request arriving during that maintenance request is much higher than later, when the maintenance task has already been firing requests for a while (due to the decreasing hazard rates, the probability of collision decreases as the system remains idle – recall Figures 3.4 and 3.5). One could, therefore, start with smaller request sizes at the beginning of a maintenance interval (when chances of a foreground request arriving are still high) and then gradually increase the request size.

We have experimented with three adaptive approaches, all of which wait for some time \(t\) before firing maintenance requests of a start size \(s\); this size is then increased with time. The exponential approach multiplies the request size by a factor \(a\) every time a maintenance request is completed without a collision occurring. The linear approach uses the factor \(a\) calculated by the exponential approach and affixes a constant \(b\) to each increase, so each new request is multiplied by \(a\) and further increased by \(b\). Using both, we can find the optimal rate of increase for our request size (we apply the exponential approach first, since it affects that rate more than the linear). We used simulations to find the constants \(a, b\) that provide the best trade-off between slowdown and maintenance throughput for each trace. We have also experimented with a simpler swapping policy, that considers only two different request sizes: when the initial wait time \(t\) is over it starts firing with the optimal request size \(s\) that achieves the average given slowdown, and then at some later point \(t'\) it switches over to the maximum request size, whose service time does not exceed the maximum allowed slowdown.

The results for our adaptive approaches are given by the dashed lines in Figure 3.8 (omitting swapping, for which we found \(t_{\text{opt}} = \infty\)). Surprisingly, we notice that none of these adaptive approaches outperforms the fixed approach where one optimal request size is picked for a given slowdown goal. Through detailed statistical analysis, we managed to identify the reason behind the poor performance of the adaptive approaches. The technique of increasing the request size only works in the presence of strongly decreasing hazard rates, i.e. over time the instantaneous probability of collision decreases. While we found this to be the case for idle time distributions in their entirety, we also found that upon “cutting off” the initial wait time, the resulting truncated distribution shows a much weaker decrease in hazard rates. In other words, the long intervals captured by the Waiting approach, are far longer than the time it takes the slowest of our adaptive approaches to reach the maximum allowed request size. Since this size will be larger than the optimal and will be reached on every captured interval, each collision will incur more slowdown than it would with the optimal size. As a result, the extra throughput comes at a cost of extra slowdown, and vice versa: when the predefined slowdown goal is considered, the corresponding throughput is lower than that for the optimal fixed approach\(^3\).

\(^3\)This is why the adaptive and 4MB Fixed approaches overlap in Figure 3.8.
3.2 Implementing a scrubber

To assess the applicability and efficiency of our scheduling approach, we would like to apply it in the context of scrubbing. Unfortunately, scrubbing has not been evaluated in implementation before, and a number of implementation choices need to be evaluated. Specifically, when implementing a scrubber, three components need to be considered: an interface directing the disk to carry out the verification of specific sectors, a module that implements the scrubbing algorithm(s), and an I/O scheduler capable of issuing scrub requests in a fashion that limits the impact on foreground traffic. We experimented with the SCSI and ATA interfaces [15], and examine their ability to issue scrub requests in Section 3.2.1. In Section 3.2.2, we briefly present the only I/O scheduler in Linux that provides I/O prioritization. Finally, we explore and evaluate different implementations of the scrubbing module in both at the user-level and within the kernel in Section 3.2.3.

3.2.1 Scrub requests in SCSI/SAS vs. ATA/SATA

Scrubbers typically rely on the VERIFY commands implemented in both the SCSI and ATA interfaces to issue scrub requests, rather than using regular disk reads. The reason is that VERIFY guarantees to verify the data from the medium’s surface (rather than reading it from the on-disk cache) and prevents cache pollution by avoiding the transfer of any data or triggering data prefetching.

Interestingly, our experiments with multiple ATA drives show that ATA VERIFY is not implemented as advertised. Specifically, we observe strong evidence that ATA VERIFY reads data from the on-disk cache, rather than the medium’s surface. Our evidence is summarized in Figure 3.9, which shows ATA VERIFY response times for different request sizes for two popular current ATA drives (WD Caviar and Hitachi Deskstar) and one SAS drive (Hitachi Ultrastar) when accessing data sequentially from the disk.

The solid lines for all models show the response times when the on-disk cache is disabled, while the dashed lines show results for when the on-disk cache is enabled. It is evident that disabling the cache affects VERIFY response times for the ATA drives but not for the SAS drive, indicating that the former do depend on the on-disk cache, rather than forcing accesses to the drive’s platter. We conclude that since ATA VERIFY is implemented incorrectly in modern ATA disks, scrubbers using them would likely pollute the on-disk cache.

Figure 3.9. Response times for different ATA VERIFY sizes
3.2.2 The Completely Fair Queueing (CFQ) I/O scheduler

In our implementation we use the Linux CFQ I/O scheduler, as it is the only open source scheduler that supports I/O prioritization. CFQ provides a separate priority class (Idle), for scheduling of background requests. To minimize the effect of background requests on foreground ones, CFQ only issues requests from the Idle class after the disk has remained idle for at least 10ms. Although this parameter is tunable, changing it in Linux 2.6.35 did not seem to affect CFQ’s background request scheduling.

3.2.3 User space, or kernel space?

While current scrubbers rely on the VERIFY commands for the reasons outlined in Section 3.2.1, there are downsides to using them due to the way they are executed by the Linux kernel. As is common with device specific functionality in Linux, a wildcard system call is used (ioctl) to pass a packet with the command and its arguments directly to the I/O scheduler, and then to the device driver for execution. However, since the kernel has no knowledge of the command that is about to be executed, such requests are flagged as soft barriers, and performance optimizations (e.g. reordering between or merging with outstanding block requests) are not applied. Since a scrubber implemented in user space has no way to avoid the performance penalty due to these scheduling decisions made in the kernel, we implement our scrubbing framework entirely in kernel space.

In Figure 3.10 we present the architecture of our kernel scrubber, implemented in the block layer of the 2.6.35 kernel [25]. The scrubber is activated at bootstrapping, matching scrubber threads to every block device in the system; this matching is updated when devices are inserted/removed, e.g. due to hot swapping. The threads remain dormant by being inserted in the CPU’s sleeping queue, until scrubbing for a specific device is activated. Internally, the scrubber implements SCSI VERIFY\(^4\), since it is not natively supported by the kernel, and provides a simple API that can be used to code new scrubbing algorithms.

---

\(^4\)Our implementation supports ATA devices through the kernel’s libATA library, which performs the appropriate translation for VERIFY.
strategies. We implemented our framework in 2700 commented LoC, and in that we coded sequential and staggered scrubbing in approximately 50 LoC each.

To send requests to the I/O scheduler and set the I/O priority for scrubber threads (if CFQ is used), we use the Generic Block Layer interface. To enable the I/O scheduler to sort scrubbing requests among other outstanding ones, every time a scrubber thread dispatches a VERIFY request we disguise it as a regular read request bearing all relevant information, such as starting LBN and request size. This information is unavailable in the vanilla kernel, since sorting is not permitted for soft barrier commands. Once the scrubbing request has been dispatched, we put the thread back to the sleeping queue, and at request completion it is awakened by a callback function to repeat the process.

Figure 3.11 shows the results from experimenting with the basic version of our kernel-level scrubber and a basic user-level scrubber. We generate a simple, highly-sequential foreground workload, with exponential think times between requests ($\mu = 100\text{ms}$) to allow for idle intervals that the scrubber can utilize. In one set of experiments we run the foreground workload and the scrubber at the same priority level (labelled Default in Figure 3.11), while in a second set of experiments we use CFQ’s Idle priority class to reduce the scrubber’s priority. For both the Idle and the Default priorities we allowed the scrubber to issue requests back to back, and for the Default priority, we also experimented with a 16ms delay between scrub requests (labelled Default 16ms in Figure 3.11), in order to allow the foreground workload to make progress. Results are shown for a Hitachi Ultrastar SAS drive, but we also experimented with a Fujitsu MAX3073RC SAS drive and got similar results.

It is evident from Figure 3.11 that when allowing the scrubber to issue requests back-to-back, both the scrubber and the foreground workload achieve significantly higher throughput in the kernel-level implementation. Also, priorities have no effect on the user-level scrubber whose requests are soft barriers, as opposed to the kernel scrubber, which is benefitting from the workload’s think time and starving it under the Default priority. When the scrubber is delayed, however, the maximum scrubbing throughput (3.9MB/s, or 64KB/16ms) is reached only by the user-level scrubber; proper prioritization limits the kernel scrubber’s throughput at 3MB/s. These results clearly motivate the use of sophisticated scheduling with I/O prioritization, if the scrubber’s impact is to be mitigated while retaining high throughput.
Chapter 3. Scheduling maintenance I/O

3.3 Getting to the bad sector at the right time

The goal of this section is to compare the performance of staggered and sequential scrubbing. While sequential scrubbing is the approach currently employed in practice, research indicates that staggered scrubbing can significantly reduce the MLET. The reason practitioners are shying away from using staggered scrubbing, is the fear that moving from sequential to more random scrubbing I/O patterns might affect the performance of the I/O system, both in terms of the scrubber’s throughout, and its impact on foreground workloads. To quantify these effects, we implemented both a sequential and a staggered scrubber within the kernel-level framework described in Section 3.2.3. We begin our experimental evaluation in Section 3.3.1 by comparing the scrubbing throughput that can be achieved by a staggered versus a sequential scrubber, and then evaluate their impact on foreground traffic, using simple synthetic foreground workloads in Section 3.3.2, and more realistic trace-based workloads in Section 3.3.3.

3.3.1 Staggered versus sequential scrubbing

The performance of both a sequential and a staggered scrubber will naturally depend on the choice of their parameters. Therefore, we begin by evaluating the effect of the scrubbers’ parameters on their performance to identify the optimal set of parameters for each scrubber.

The only tunable parameter of a sequential scrubber is the request size \( S \) of each scrubbing request, a parameter shared with the staggered scrubber. Our first goal in this section is to distinguish the range of request sizes that can achieve high scrubbing throughput. We measured the response time of SCSI Verify for different request sizes for two SAS drives (Hitachi Ultrastar 15K450 300GB and Fujitsu MAX3073RC 73GB) and one SCSI disk (Fujitsu MAP3367NP 36GB), and we found that for requests \( \leq 64\text{KB} \), response times remain almost constant for all models as shown in Figure 3.12. As a consequence, we henceforth experiment only with request sizes starting at 64KB, and report the throughput achieved by each scrubber. The results for the two SAS drives are shown in the two solid lines in Figure 3.13a, suggesting that the largest scrubbing request size should be preferred, from 64KB to 4MB.

In the case of the staggered scrubber there is an additional parameter: the number of regions that...
Figure 3.13. Impact of scrubbing parameters on sequential and staggered scrubbing performance.

The disk is divided into. Recall that a staggered scrubber separates the disk into \( R \) regions, based on a predefined region size, each of which gets scrubbed in \( \left\lceil \frac{S}{R} \right\rceil \) rounds. In the first round, the first \( S \) bytes are verified from each region, then the \( S \) after those, and so on. To experiment with the region size, we ran the staggered scrubber on our two SAS disks using 64KB requests (the presented trend holds regardless of request size), dividing the disks in up to 512 regions. The solid lines in Figure 3.13b represent the staggered scrubber’s throughput for the two drives as a function of the number of regions. We observe that the throughput of the scrubber continuously increases as the number of regions increases from two to 512 (in the case of one region, the staggered scrubber’s actions are identical to a sequential scrubber). To answer our original question, which is how the performance of a staggered scrubber compares to that of a sequential scrubber, the dashed lines in Figure 3.13b represent the throughput achieved by a sequential scrubber (using 64KB requests). Interestingly, for more than 128 regions we find that the staggered scrubber performs equally well or better than the sequential one. We have observed this trend for drives of varying capacities, which leads us to assume that it is independent of the disk’s capacity. To verify that this trend holds for larger request sizes, we have also plotted the throughput of a staggered scrubber as a function of the request size (while fixing the number of regions to 128) in Figure 3.13a. Since work on the impact of staggered scrubbing on the MLET shows that the number of regions has a relatively small impact on MLET [163], we recommend using small region sizes, compared to the disk’s capacity. To obtain conservative results, though, we fix the number of regions to 128 in subsequent experiments.

The fact that staggered can outperform sequential scrubbing may seem counter intuitive. However, since VERIFY avoids transferring data to the controller or on-disk cache, when a sequential scrubbing request is completed, the next one will have to be serviced from the medium’s surface. At the same time, the head has moved further along the track, while the VERIFY result was propagated to the controller. Hence, when the next VERIFY is initiated, the head needs to wait for a full rotation of the platter until it reaches the correct sector. For the staggered scrubber, this describes only the worst case. However, when the regions are too large (less than 64 in total), the overhead of jumping between regions dominates the overhead caused by rotational latency. We have validated this hypothesis experimentally by increasing the delay between subsequent scrubbing requests, by intervals smaller than the rotational latency. As expected, only the staggered scrubber was harmed by such delays.
Chapter 3. Scheduling maintenance I/O

3.3.2 Scrubbing impact on synthetic workloads

Next, we evaluate the performance impact of scrubbing on two simple synthetic foreground workloads. The first is a workload with a high degree of sequentiality: it picks a random sector and reads the following 8MB using 64KB requests. Once the chunk’s last request is serviced, it proceeds to pick another random sector and start again. The second is a random workload, which reads random 64KB chunks of data from the disk. For both workloads we insert an exponentially distributed think time between requests, with a mean of 100ms, to allow for idle intervals that the scrubber can utilize. In all cases, we send requests directly to the disk, bypassing the OS cache.

We experiment with both the staggered and the sequential scrubber using 64KB scrub requests, which represent the best case in terms of collision impact by imposing minimal slowdown (recall Figure 3.12). We schedule scrub requests in two commonly used ways. In the first case we issue scrub requests back-to-back through CFQ, using the Idle priority to limit their impact on the foreground workload. In the second case we use the Default priority and limit the scrubber’s rate by introducing delays between scrub requests, ranging from 0-256ms (anything larger scrubs less than 320GB bi-weekly).

The results for the sequential workload are shown in Figure 3.14a. We observe that the highest...
combined throughput for the workload and the scrubber is achieved with CFQ (where the scrubber submits requests back-to-back); however, this comes at a significant cost for the foreground application: a drop of 20.6% in throughput, compared to the case when the foreground workload runs in isolation. When inserting delays between scrub requests, rather than using CFQ to limit the impact of the scrubber, we see that for large enough delays ($\geq 16$ ms) the throughput of the foreground workload is comparable to that without a scrubber. However, in those cases the throughput of the scrubber is greatly reduced, from 9 MB/s under CFQ to less than 3 MB/s with a delay of more than 16 ms. As a secondary result, we note that again there is no perceivable difference between the staggered and the sequential scrubber for sufficiently small regions (here, the number of regions used were 128).

We observe similar results when we scrub against the random workload in Figure 3.14b: to achieve application throughput comparable to the case without a scrubber in the system, large delays are required that cripple the scrubber's throughput. Note that random workloads induce additional seeking, decreasing the scrubber's throughput.

### 3.3.3 Scrubbing impact on real workloads

To experiment with more realistic workloads, we replayed a number of real I/O traces from the HP Cello and MSR Cambridge collections shown in Table 3.1. Figure 3.15 shows the impact of our scrubbers on one of these (more realistic) workloads by plotting the cumulative distribution function of the response times of application requests. For better readability we only include results for four different cases: no scrubber; back-to-back scrub requests scheduled through CFQ’s Idle priority class; and scrub requests with delays of 0 ms and 64 ms between them.

Again, we observe results very similar to those for the synthetic workloads. Using back-to-back scrub requests, even when lowering their priority through CFQ, greatly affects the response times of foreground requests. On the other hand, when making delays between scrub requests large enough to limit the effect on the foreground workload (64 ms), the throughput of the scrubber drops by more than an order of magnitude (the scrubber’s throughput for each experiment is included in the legend of Figure 3.15). As before, the results are identical for the staggered and the sequential scrubber. These results motivate us to look at more sophisticated ways to schedule scrub requests.
3.3.4 Putting it all together

In Section 3.1 we made some interesting observations for optimizing the background scheduling of maintenance requests in I/O schedulers. First, we found that a simple approach based on waiting outperforms more complex ones based on auto-regression. Second, we found that picking one fixed request size for the maintenance task, rather than adapting it within an idle interval, is sufficient. This allows for a simple policy with only two tunable parameters: the maintenance request size and the wait time threshold. We further found that for a given slowdown target these two parameters can be determined relatively easily, based on a short I/O trace capturing the workload’s periodicity, and simulations guided by binary search. The simulations can be repeated to adapt the parameter values if the workload changes substantially.

<table>
<thead>
<tr>
<th>Disk</th>
<th>Average Slowdown</th>
<th>Throughput</th>
<th>Threshold</th>
<th>Request Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>HPc6t8d0</td>
<td>1.0 ms</td>
<td>38.75 MB/s</td>
<td>205 ms</td>
<td>1280KB</td>
</tr>
<tr>
<td>(Waiting)</td>
<td>2.0 ms</td>
<td>50.53 MB/s</td>
<td>88.1 ms</td>
<td>1536KB</td>
</tr>
<tr>
<td></td>
<td>4.0 ms</td>
<td>61.51 MB/s</td>
<td>32.4 ms</td>
<td>2048KB</td>
</tr>
<tr>
<td>CFQ</td>
<td>2.3 ms</td>
<td>8.25 MB/s</td>
<td>10 ms</td>
<td>64KB</td>
</tr>
<tr>
<td>HPc6t5d1</td>
<td>1.0 ms</td>
<td>68.12 MB/s</td>
<td>632 ms</td>
<td>3072KB</td>
</tr>
<tr>
<td>(Waiting)</td>
<td>2.0 ms</td>
<td>72.83 MB/s</td>
<td>340 ms</td>
<td>4096KB(^6)</td>
</tr>
<tr>
<td></td>
<td>4.0 ms</td>
<td>73.97 MB/s</td>
<td>247 ms</td>
<td>4096KB(^6)</td>
</tr>
<tr>
<td>CFQ</td>
<td>2.6 ms</td>
<td>9.25 MB/s</td>
<td>10 ms</td>
<td>64KB</td>
</tr>
<tr>
<td>MSRsrc11</td>
<td>1.0 ms</td>
<td>73.54 MB/s</td>
<td>22.3 ms</td>
<td>3072KB</td>
</tr>
<tr>
<td>(Waiting)</td>
<td>2.0 ms</td>
<td>75.77 MB/s</td>
<td>15.3 ms</td>
<td>4096KB(^6)</td>
</tr>
<tr>
<td></td>
<td>4.0 ms</td>
<td>76.40 MB/s</td>
<td>12.6 ms</td>
<td>4096KB(^6)</td>
</tr>
<tr>
<td>CFQ</td>
<td>7.7 ms</td>
<td>13.19 MB/s</td>
<td>10 ms</td>
<td>64KB</td>
</tr>
<tr>
<td>MSRusr1</td>
<td>1.0 ms</td>
<td>62.49 MB/s</td>
<td>10.8 ms</td>
<td>1472KB</td>
</tr>
<tr>
<td>(Waiting)</td>
<td>2.0 ms</td>
<td>71.24 MB/s</td>
<td>39.8 ms</td>
<td>3072KB</td>
</tr>
<tr>
<td></td>
<td>4.0 ms</td>
<td>71.58 MB/s</td>
<td>18.5 ms</td>
<td>4096KB(^6)</td>
</tr>
<tr>
<td>CFQ</td>
<td>106 ms</td>
<td>13.44 MB/s</td>
<td>10 ms</td>
<td>64KB</td>
</tr>
</tbody>
</table>

Table 3.3. Fixed waiting approach results for different disk traces

We have simulated our scheduling policy in the case of scrubbing. Table 3.3 summarizes the throughput a scrubber can accomplish for four of our traces and three average slowdown goals of one, two and four msec, respectively. The table also provides the wait time threshold and the request size that were used to achieve those throughputs. To put these results into perspective, we have also simulated CFQ’s policy for scheduling background work, albeit for 64KB requests. We find that our scrubber achieves significantly less slowdown (up to 3 orders of magnitude for busier traces) for up to 64x larger requests, yielding significantly more throughput per ms of slowdown. CFQ comes (somewhat) close to our approach only when its fixed threshold (10 ms) happens to align with the workload.

\(^6\)The maximum slowdown allowed (50.4 ms) limits the request size at 4 MB. If that restriction is relaxed, this run can achieve higher throughput.
Chapter 4

Opportunistic storage maintenance

Maintenance tasks raise challenges for storage systems because they access a significant amount of data that does not easily fit in memory. For example, enterprises typically run full backups weekly and often more frequently, as we showed in Chapter 2. Similarly, anti-virus scans in virtual machines cause I/O storms [212]. These tasks can interfere with foreground applications, which we call the workload, causing significant impact on their performance. Thus, administrators have to carefully schedule maintenance tasks during idle times. However, long idle times may not be available, especially with increasing data storage needs. For instance, as enterprises are moving to the cloud, data sharing occurs across time zones, and much higher consolidation ratios are observed in storage systems. As a result, workloads are losing their traditional diurnal characteristics that guarantee predictable idle periods, making it harder to meet maintenance goals. A recent survey of 500 CIOs of medium scale organizations confirms this trend, showing that 40% of Microsoft SMB backups fail to complete within their scheduled window [237, Chart 13]. Another survey of 1200 IT professionals shows that 33% of backups routinely miss their window, while only 28% always complete on time [108, p. 3]. Ironically, over 50% of IT professionals believe that someone could lose their job if critical data was lost after a disaster [61].

Existing approaches for minimizing the impact of maintenance tasks focus on I/O scheduling, taking device characteristics into account. On hard disks, maintenance is piggybacked on workload requests [62] or performed during the seek time and rotational latency between workload requests [144, 226]. These approaches require detailed device performance characteristics to be determined, which is non-trivial in modern disks [126], and even more complicated for SSDs [5]. To be effective, they also require applications to a-priori specify the I/O requests they plan to issue. Furthermore, they need complex mechanisms for handling inter-block dependencies [226], as discussed in the next section. Other approaches consider the characteristics of the idle periods, such as the approach presented in Chapter 3.

We propose a novel maintenance approach that prioritizes processing of data cached in memory, which complements I/O scheduling approaches already present in the literature. Data may be cached as a result of other maintenance tasks requesting it, or due to overlapping foreground I/O activity. This approach reduces the impact of maintenance in two ways. First, maintenance tasks can implicitly collaborate with each other. For example, during a full logical file system backup, data layout reorganization (such as defragmentation or garbage collection) can be performed with no additional reads. Second, data that has recently been accessed can be provided to a maintenance task. For example, a block modified by the workload can be used by an incremental backup task, avoiding an additional read. This I/O reduction
helps maintenance tasks complete their work within their scheduled windows.

Our key insight is that maintenance work can be reordered. Maintenance tasks typically process items, such as blocks or files, in some predefined order, but they are not dependent on this order. For instance, a system administrator might require a backup to complete within a few hours. Within that period, backup of different files can be reordered without affecting the system’s reliability guarantees.

We present Duet, a storage maintenance framework that provides hints to tasks about page-level events, such as a page being added, or modified in the page cache. Tasks use these hints to process their items out-of-order, which we call opportunistic work. For example, tasks can prioritize processing of files that have more pages in memory. Page-level events provide fine-grained information, helping support items of various granularities such as blocks, files, extents etc. Duet helps track completion of opportunistic work so that it is not repeated by the task.

An important goal of our work is to provide a simple programming model that minimizes changes to the tasks. We have modified five existing maintenance tasks to work with Duet. In the kernel, we have modified the scrubbing, backup, and defragmentation tasks in the Btrfs file system [190], and garbage collection in the F2fs log-structured file system [131]. These tasks required changes to fewer than 150 lines of code (LoC) each, or 2-10% of the original code. At the application level, we have modified the rsync application [231, 232], which required changes to 300 LoC, or 0.67% of its code.

We show that maintenance tasks using Duet can complete faster because they perform fewer I/O operations. This reduction depends on the amount of overlapping data accessed by the various maintenance tasks and the workloads. Tasks using Duet implicitly collaborate with each other, and are able to complete within their scheduled windows, even on busy devices. For example, when scrubbing, backup, and defragmentation are run concurrently with a workload that keeps the device busy 50% of the time, Duet-enabled tasks perform their work four times faster than the original tasks.

Our work makes three contributions. We identify that maintenance tasks can process data items in arbitrary order without affecting correctness. We provide a simple programming model that enables these tasks to perform opportunistic processing based on data cached in memory. We apply this model to several kernel and user tasks, with minimal task changes, and show the performance benefits obtained.

The rest of the chapter describes our approach in more detail. Section 4.1 presents our opportunistic work model, and Section 4.2 describes the Duet framework that implements this model. Then, Section 4.3 explains how we modified five maintenance tasks to use Duet, and Section 4.4 demonstrates the effectiveness of the Duet-enabled tasks. Finally, in Section 4.5 we present preliminary work that extends our opportunistic work model to distributed computational frameworks, and as a mechanism for providing finer granularity information to applications that depend on file notification frameworks today.

### 4.1 The Opportunistic Work Model

Our aim is to enable one or more maintenance tasks to execute concurrently, with minimal impact on the foreground workload. To do so, we leverage the property that maintenance work can usually be reordered. Maintenance tasks typically process items, such as data blocks or files, in a predefined order but they are not dependent on this order. Our Duet framework provides tasks with hints about cached data. Tasks can use these hints to opportunistically process cached data out-of-order, reducing the total I/O required to meet their goals.
Our work has two sub-goals. First, to encourage adoption, we would like to provide a simple programming model that minimizes changes to the tasks. This requirement introduces several challenges. We would like to reuse existing maintenance tasks rather than writing them from scratch. These tasks are developed independently, so rewriting each task to explicitly collaborate with every other task in the system is unreasonable, both due to the large number of possible combinations of tasks, and the effort needed to add new tasks. We should also not require tasks to specify maintenance I/O a-priori, because it is too onerous on the developer, and constrains the ability to adapt tasks to changes in the system. Finally, tasks operate at several granularities (e.g. blocks, files, extents, segments), and we need to easily support all of them. Our second goal is to design a framework that supports a variety of maintenance tasks and scales with the number of tasks running on the system.

Next, we provide an overview of Duet and how it meets these goals. The design of Duet is described in more detail in Section 4.2.

4.1.1 Overview of Duet

Duet hooks into the page cache and provides notifications about page-level events to maintenance tasks, such as a page being added, removed, dirtied or flushed (written back to storage) from the page cache. We leverage page-level events because data is cached at page-size granularity when it is read from and written to storage. Tasks use the Duet API to poll for these events at appropriate times, such as before each item is processed. We provide a polling interface because it avoids complications with handling asynchronous events. The tasks then use their own criteria to decide how and when to act upon the events. The result is that tasks perform out-of-order processing of cached items, reducing I/O operations. Duet helps track work completion so that tasks operate once on each item, either opportunistically or during normal operation.

While Duet events occur at the page granularity, a task, such as defragmentation, may operate at a different granularity, such as extents. For example, the task may require all pages of an extent to be in memory before defragmenting it, but the task will receive notifications when any of these pages are brought into memory by other applications. Instead of encumbering Duet with inter-page dependencies, the Duet events help tasks keep track of data available in memory, so that they can perform opportunistic processing. For example, the defragmentation task can use the Duet events to build a priority queue based on the extents with the most pages in memory. It can then use this queue to prioritize its processing, requesting from storage any additional pages of that extent that are needed to complete its operation. This approach avoids pinning pages in memory and the related issues that arise under memory pressure [226].

Algorithm 1 shows a simple example of a Duet file task, based on the Defragmentation and Rsync file tasks that we adapted for Duet (see Table 4.3). The *sid* parameter is a session id, and the Duet API calls are shown in bold, as described in Section 4.1.2. The original task runs a loop, calling `pick_next_file` (line 6) to choose files in some predetermined order, and then invoking `process_file` (line 31) to process the chosen file. The `handle_queued` (line 4) function performs opportunistic processing. It uses `prioqueue_update` (line 13), which fetches pending page events from Duet (line 22) and updates a priority queue of inodes (line 24). The priority queue is sorted by some task-specific criteria, such as the number of pages the inodes have in memory. Next, the `handle_queued` function dequeues the highest priority inode (line 14) and processes it opportunistically. The `handle_file` code is modified to check whether inodes have been processed already and to mark them as processed. This processing repeats,
Algorithm 1. An example Duet task.

aggressively fetching page events again because the page cache may have changed, until there are no more items in the priority queue.

Our approach helps meet the goals described earlier. It does not require tasks to specify their pending work to Duet a-priori. In particular, a task can change the work it performs while it is running, without informing Duet. For example, a defragmentation task in a copy-on-write file system can simply ignore an overwritten file that it was planning to defragment. Duet also does not need to know about subtle dependencies that exist in the task, such as a page A needing to be processed before another page B. With our best-effort approach, the task can simply ignore inopportune events, such as page B being available in memory before page A. While our approach requires tasks to be modified, they do not need to explicitly collaborate or be aware of each other.

4.1.2 Duet API

Duet supports maintenance tasks operating at either the block or the file system layer. Block layer tasks, or block tasks, operate on data blocks, while file system layer tasks, or file tasks, are aware of files and
Chapter 4. Opportunistic storage maintenance

```c
int duet_register(path, notification_mask)
int duet_deregister(session_id)
int duet_fetch(session_id, item_array, count)
int duet_check_done(session_id, item_id)
int duet_set_done(session_id, item_id)
int duet_unset_done(session_id, item_id)
int duet_get_path(session_id, inode_num, path)
```

Table 4.1. The Duet API

<table>
<thead>
<tr>
<th>Event</th>
<th>State Change</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Added</td>
<td>Exists</td>
<td>Page added in cache</td>
</tr>
<tr>
<td>Removed</td>
<td>¬ Exists</td>
<td>Page removed from cache</td>
</tr>
<tr>
<td>Dirtied</td>
<td>Modified</td>
<td>Dirty bit set for page</td>
</tr>
<tr>
<td>Flushed</td>
<td>¬ Modified</td>
<td>Dirty bit cleared for page</td>
</tr>
</tbody>
</table>

Table 4.2. Event and State-based Notifications

Table 4.1 shows the Duet API. Maintenance tasks call `duet_register` to start using Duet. In the `path` parameter, block tasks specify a device file, while file tasks specify a directory (which we call the `registered` directory). The `notification_mask` consists of the types of events that are of interest to the task, as shown in Table 4.2 and explained later in the section. The `duet_register` call returns a `session_id` that is used in the rest of the Duet calls. The task ends the session when its work is complete by calling `duet_deregister`, which releases all Duet session state.

The heart of the Duet API is the `duet_fetch` system call that provides notifications to tasks about page-level events. These notifications are returned in `item_array`, up to a maximum of `count`, similar to the `read` system call. The fetch call returns any events that have occurred but have not yet been returned by previous calls to fetch. Block tasks receive notifications for any page-level events occurring on the device, while file tasks receive them for page-level events on all the files or directories located within the registered directory and its sub-directories.

Table 4.2 shows that tasks can register to be notified about page events or changes in page state. Event notifications are triggered when a page is added, removed, modified, or flushed from the cache. State notifications are emitted when the existence or modification status of a page changes in the page cache. For example, if a task registers for Exists notifications, and a page is removed and re-added between two consecutive fetch operations, then the page is considered to have reverted back to the same state, i.e. it exists in the cache, and an event is not generated on the next fetch call.

An item in `item_array`, returned by `duet_fetch`, consists of a tuple `(item_id, offset, flag)`, that corresponds to a given page. For block tasks, the `item_id` is the block number, while for file tasks it consists of the inode number uniquely identifying a file or directory. The `offset` is only used for file tasks, and corresponds to the logical offset within the file. The `flag` field consists of six bits, one for each event and state notification type shown in Table 4.2. This field identifies only the page events that have not yet been made available to the task via fetch operations. For example, suppose a page is added, a fetch operation occurs, and then the page is removed. The next fetch call will return an item for the
page with only the removed bit set in the item flag, informing the task that the page has been removed.

To track whether items have been processed, tasks use the \texttt{duet\_\_done} calls (Table 4.1), as described in more detail in Section 4.2. Finally, file tasks use \texttt{duet\_get\_path} to translate an item’s inode to a path relative to the registered directory. We provide this call, rather than returning file paths in \texttt{duet\_fetch} for two reasons. First, tasks will generally invoke \texttt{duet\_get\_path} once per file, but \texttt{duet\_fetch} is called much more frequently because it operates at page granularity. Second, \texttt{duet\_get\_path} serves as the \textit{truth} for our page cache hints [129]. When it fails, it indicates that the file is no longer cached, allowing tasks to back out of opportunistic processing that may not be worthwhile. Section 4.3 shows how tasks use these primitives.

### 4.1.3 Discussion

Our initial design for Duet was event-driven, with tasks being informed as I/O requests moved through the page cache, file system, and block layers in the storage stack. While this approach provided comprehensive control over storage processing, we found it tedious to implement and use. The implementation challenges arose from requiring changes across the storage layers, while in practice we found that page-level events are sufficient for maintenance tasks.

The event-driven approach is hard to use because existing maintenance tasks are not designed to process data made available at arbitrary times, either because of synchronization issues with on-going processing, or because of dependencies with unavailable data. Our current polling-based approach is easier to retrofit in maintenance tasks because it allows them to poll for events and perform out-of-order processing at suitable times. However, polling makes no guarantees of data availability between fetching and processing an item. For this reason, we take advantage of the page cache, which provides sufficient time to detect and exploit synergies between tasks. A side benefit is that our current approach does not require any file-system specific changes.

While Duet does not change the file access control model because \texttt{duet\_fetch} doesn’t provide any file data, it can leak information about pages in memory. For block tasks, we require them to be able to access their block device. For file tasks, we use file permissions to return events for files that are accessible to the task, but we currently do not take path-based access control into account.

Note that tasks using direct I/O will not benefit from Duet because they bypass the page cache. However, the maintenance tasks that we have examined do not use direct I/O. Tasks such as databases use direct I/O to handle data caching themselves, and we plan to apply the Duet API to such caches at the user level. Similarly, Duet currently provides no support for hard links. Specifically, if a file that belongs to a task’s registered directory is accessed through a hard link, Duet will not report related events to the task. In the current implementation, Duet matches inode numbers to directory entries that are loaded in memory. As a result, aliases for the file that are available through hard link will not be visible unless they have been accessed recently. This scenario is possible today in the context of consumer-level backup applications, such as Apple’s TimeMachine [228], where incremental backups are constructed using hard links to previous backups. While hard link usage is a controversial topic and generally discouraged due to security concerns [37, 75], we are working on an efficient solution to provide proper support for applications that require it.

Finally, there are some similarities between Duet and the Linux Inotify mechanism [105, 122] that reports file-level accesses to applications. Inotify is used by applications such as the file manager, desktop search utilities, and for file synchronization (e.g. Dropbox, Google Music Manager). While Inotify focuses
on file-level accesses, Duet is designed to track file data in memory. As a result, Duet provides page-level information, which is finer grained than Inotify’s file-level information, allowing better prioritization for out-of-order processing. Similarly, Duet provides information about when data is flushed and evicted. Unlike Duet, Inotify does not support watching directories recursively, so adding watches to each subdirectory can take significant time for large directories, and is race prone. On the other hand, Duet does not inform tasks about file metadata changes (e.g. permissions, extended attributes).

4.2 The Duet Framework

This section describes the design and implementation of the Duet framework.

4.2.1 Framework Design

Duet hooks into the page cache modification routines and gets control when a page is added or removed from the page cache, or when a page is marked dirty or flushed. When these page cache events occur, Duet is passed a page descriptor and an event type, such as a page being added, removed, etc. Duet traverses the list of sessions, examining the notification mask registered by each session to determine whether the event type is of interest. If so, we determine whether the page is relevant to the session by checking whether it belongs to the correct device for a block task, or whether it lies within the registered directory for a file task. If the page is relevant, and has not been marked done, we update its set of pending events.

Duet maintains an item descriptor for each relevant page with pending events. The item descriptor contains the same information that a fetch call returns, i.e. an item of type \( (item_id, offset, flag) \), as described in Section 4.1.2. An item descriptor contains pending events when one or more bits are set in its flag field. A fetch call returns item descriptors with pending events, and marks the descriptors up-to-date by clearing their flag fields.

When a session is registered, we scan the page cache and initialize an item descriptor for each relevant page. The flag field is set to indicate that the page is present (and possibly dirty). This scan serves two purposes. First, it is required for state notifications to be generated correctly, such as the Exists state from Table 4.2. Second, a scan allows a task to immediately take advantage of pages in the page cache (by invoking fetch after session registration).

While item descriptors help track pending events, Duet can also benefit from tracking relevant pages so that events can be generated more efficiently. This is a performance, rather than a correctness requirement, allowing Duet to avoid storing the extra events. As we demonstrate later (Figure 4.11), the number of pending events can slightly increase Duet’s overhead. To ascertain page relevance, Duet maintains per-session state consisting of either one or two bitmaps. For block tasks, Duet maintains a single done bitmap that tracks completed work. The bitmap stores one bit for each block on the device. Tasks use the duet_*_done functions shown in Table 4.1 to check, set, and reset bits in the bitmap. When the task marks a block as done, the associated item descriptor is marked in the bitmap. Future events on the page are then ignored until the done bit is unset by the task. Although tasks decide when some work is complete and can track completed pages themselves, informing Duet avoids tracking and generating events for completed pages.

For file tasks, Duet maintains two bitmaps, done and relevant. Both bitmaps store a bit for each inode (i.e., for each file or directory) in the file system that contains the registered directory. When a
file is marked in the done bitmap, the item descriptors for all the associated pages of the file are marked up-to-date and future events on the file are no longer tracked. We use file-level marking because these tasks operate at file granularity.

File tasks are only interested in files or directories located within the registered directory. We use the relevant bitmap to ensure that fetch only returns events on these relevant objects. When any page of a file (or directory) is added to the page cache for the first time, we traverse its path backwards to detect whether the file lies within the registered directory. This check would be expensive if applied on every page access, so after the first access we use the relevant bit to determine whether the corresponding inode is relevant. If the inode is not relevant, we immediately mark the file as done, thus avoiding tracking the file pages or generating any events for the file in the future. Otherwise, we mark the relevant bit, and consult it on page cache events before generating notifications for the file.

Duet also needs to handle files and directories being moved into, or out of, the registered directory. We detect that a file is moved into the registered directory at the VFS layer and initialize item descriptors for all pages of the file in a manner similar to the page scan performed during session initialization. When a file is moved out of the registered directory, we set the Removed bit and clear the Exists bit for all existing pages of the file, marking the file as done. After the next fetch, Duet will ignore the file.

Directory renames are trickier, because they require handling all files and directories under the renamed directory. Duet deals with directory renames by resetting the relevant and done bitmaps for all files other than the files that have already been processed, i.e. have both bits set. This approach avoids the need to traverse the renamed directory, and it guarantees that tasks will not receive unnecessary events for processed files. However, it requires rechecking file relevance when the files are accessed again.

### 4.2.2 Implementation

Our implementation of Duet consists of three components: a Linux kernel module, hooks in the Linux page cache, and a library for user and kernel tasks, implemented for Linux 3.13. Our implementation consists of 1700 lines of code for the first two components, and 1000 lines for the library.

While the item descriptors of different sessions are logically independent, we reduce memory requirements by keeping a single item descriptor per page for all sessions. The merged item descriptor consists of the item_id, offset, and an N-byte array for storing the flag fields for up to a maximum of N concurrent sessions. This maximum value can be configured at module load time. With this implementation, we allocate a descriptor when any session has pending events on the page, and deallocate it when no session has pending events on the page.

The merged descriptor implementation allows using a more efficient single, global hash table to look up the descriptors. We use the item_id and offset as the hash key, and then use the session id (which ranges between 0 and \((N-1)\)) to index into the flag array.

Note that an item descriptor with pending events will remain allocated even if the corresponding page is deallocated from the page cache. For tasks that only subscribe to event notifications, the descriptor is only deallocated when it is marked up-to-date by a fetch call. Thus, item descriptors can grow over time if a task does not issue fetch calls. To counter denial of service, we limit the number of item descriptors per session and drop new events when this limit is reached. Note that this issue did not affect our Duet tasks because they invoke fetch calls many times per second, as explained in Section 4.4.4.

---

\(^1\)This operation is relatively efficient in our Linux implementation, which maintains a directory entry cache that pins in memory the directory path leading to a file page.
Table 4.3. Tasks adapted to use Duet and their characteristics

When a task registers for state notifications (e.g. Exists), a page can also be marked up-to-date when the corresponding events cancel each other, such as when a page is added and subsequently removed from the cache. As a result, the maximum number of item descriptors are bounded by \((2 \times \text{max. number of pages in cache})\). This bound would be reached if all existing pages are relevant, and they are removed and subsequently replaced by new pages between fetch calls. With this bound, events are never dropped.

We use a red-black tree to dynamically allocate portions of the relevant and done bitmaps, to represent ranges that have marked bits, and deallocate them when all their bits are unmarked or when the session terminates. This limits memory consumption when tasks are interested in small, localized chunks of a device or file system.

Duet allows synergies to be detected between block and file tasks. For example, a block device could be mounted as a file system. In this case, we would like to provide any block task operating on the device with page events for files accessed by file tasks, or applications operating on the file system. However, the translation of a page’s file offset to a block number is a filesystem-specific operation, and Duet is filesystem agnostic. Fortunately, many file systems in Linux (e.g. Btrfs, Ext2/3/4, XFS, F2fs) implement this translation through the FIBMAP ioctl [141, 234]. We use this functionality, when implemented, to inform block tasks of file-level accesses on the same device. A similar API exists in Windows [188]. In the event that a page does not correspond to a block yet (e.g. due to delayed allocation [56]), the page is left to be returned by a later fetch operation.

Duet ignores pages not backed by files because they are not useful for maintenance tasks. It further provides both file and directory pages to file tasks, although our current tasks ignore directory pages. Finally, the Duet library is used by both in-kernel and user-level tasks. It implements a priority queue for storing Duet events that are fetched using the Duet API. Through this library, tasks can access the Duet API (Table 4.1), and the priority queue primitives shown in Algorithm 1. Our current implementation uses a red-black tree for the priority queue.

4.3 Applications

This section describes how applications use the Duet framework. Table 4.3 shows three block and two file tasks that we modified to work with Duet. Specifically, we modified the existing in-kernel scrubbing,
backup, and defragmentation utilities available in the Btrfs copy-on-write file system [190]. We have also modified the garbage collector used by the log-structured F2fs file system [131]. These modifications were made in Linux 3.13. Finally, we have changed version 3.1.1 of the Rsync user-level application [231, 232]. We found that the overall structure of these applications was similar: they consist of a loop in which items are processed in a predefined order. Table 4.3 describes the order in which each application normally processes items. Since the operation applied on each item is idempotent, we can insert code within the loop that checks whether there are items with cached pages and processes those before falling back to the predefined processing order. To avoid potential issues with repeated work, the code responsible for processing a single item is wrapped around Duet’s primitives for bitmap setting/checking, similar to the example in Algorithm 1. Table 4.3 outlines the changes we had to make to each task at a higher level. Overall, the changes required per task did not exceed a couple hundred lines, and they are described in more detail in the following subsections.

4.3.1 File System Scrubbing

To protect against data loss due to silent data corruption [28, 29, 127], commercial storage systems rely on scrubbing [163, 202]. Recall from Chapter 3, that a scrubber is a background process periodically scanning data and verifying its correctness using checksums. While scrubbing is commonly performed at the block layer, the Btrfs scrubber operates within the file system protecting against a wider variety of errors [127]. In Btrfs, a checksum is stored for every file system block, updated on a block write, and verified on a block read to ensure that applications receive correct data. The scrubber reads all allocated file system blocks on a given device sequentially and verifies them against their checksums for correctness.

Our opportunistic scrubber relies on the semantics of the file system’s read and write operations to reduce maintenance work, while providing the same reliability guarantees as the original scrubber. The opportunistic scrubber receives notifications when a page is Added or Dirtied in the page cache. When a page is added, we mark the relevant block number as scrubbed, since Btrfs verifies data correctness during the read operation. On the other hand, checksums are not verified on a write request, so we unmark the bit for the dirty page block, indicating that the new checksum needs to be re-verified.

4.3.2 Snapshot-based backup

Btrfs is a copy-on-write file system that supports taking fast, file-system snapshots. All data and metadata in the snapshot is shared with the live file system until blocks are updated in the live system. Btrfs provides backup tools that allow taking a consistent backup using a read-only snapshot.

Our opportunistic backup tool exploits copy-on-write sharing because read operations to the live data may access data shared with the snapshot that is being backed up. By registering the backup session with Duet for the Exists notifications, we are informed of pages that currently exist in the page cache and their corresponding block numbers.

To perform opportunistic processing, the backup tool locks a page, checks that it is not dirty and then copies it to a private buffer. Next, it checks that the page has not been modified since the snapshot using back-references in Btrfs, and then unlocks the page. Finally, the data from the private buffer is sent out-of-order to the backup storage.
4.3.3 File Defragmentation

Due to the copy-on-write nature of Btrfs, any write to a file stores the new data in unused blocks. This layout reorganization causes fragmentation, especially for small random writes. Btrfs allows defragmenting a file by merging small extents with logically adjacent ones. The existing Btrfs tool allows defragmenting one file at a time at the user level. We have reimplemented this tool in the kernel to speed up defragmentation for multiple files and directories. Our in-kernel implementation uses metadata prefetching during namespace traversal, speeding it up by a factor of 10. We use this implementation as the baseline for our experiments.

Our opportunistic defragmenter monitors Exists notifications to track files that have data in memory, and prioritizes those files with the highest fraction of pages in memory compared to their size, similar to the example in Algorithm 1.

4.3.4 Garbage Collection

F2fs is a log-structured file system [193], designed to perform well on flash storage [131]. F2fs groups blocks in segments. When a block is updated, it is appended to the log, and its previous version becomes invalid (in some segment). Segments with many invalid blocks are cleaned by a background garbage collector that copies the remaining valid blocks in the segment to the log, freeing the segment for logging future writes. The garbage collector prioritizes segment cleaning using a cost function based on the amount of data that needs to be moved and the segment’s age. It runs when the device is idle, cycles through 4096 segments at a time (instead of all segments on the device), and cleans one segment with the minimum cost (i.e. the most invalid blocks).

Our opportunistic garbage collector modifies the cost function to account for the number of valid blocks of a segment that are cached because these blocks save read operations. During cleaning, a segment’s blocks are synchronously read from storage, and marked dirty in memory for asynchronous writeback. We conservatively weigh both read and write operations equally, and change the number of blocks that need to be moved from \( \text{valid blocks to cached blocks  \text{-} valid blocks/2} \) in the cost function.

The garbage collector also monitors Flushed notifications. When a block is flushed to disk, it is mapped to a new segment, and its copy in the old segment is invalidated. We track block addresses mapped to in-memory segments in a hash structure, so that when a flush happens we have information on which segment the block moved to, and which segment was the old one, in order to decrement the count of the latter. On a flush event, we adjust the in-memory counters for both the old and new segments. Interestingly, the notion of completed work does not apply to the garbage collector because a segment can always become dirty again, and so the Duet done primitives are not used.

4.3.5 Rsync Application

Rsync is a widely-used user-level tool for synchronizing the contents of a source and a destination directory. It uses data checksums to find differences between source and destination files, sending only the updated data blocks. Rsync consists of three processes that communicate via sockets and pipes. The sender process is responsible for traversing the directory hierarchy at the source and sending the file metadata to the receiver process, which passes it to the generator process. The generator calculates file checksums and sends them to the sender, which then generates its own checksums to detect updated blocks. Finally, updated data is sent to the receiver, which updates the destination files.
Chapter 4. Opportunistic storage maintenance

The opportunistic rsync uses the Exists notifications to track files that have data in memory, prioritizing files with the highest number of pages in memory, similar to the example task shown in Algorithm 1. It ensures that the metadata for a file is sent once in the first step, either opportunistically or during normal operation.

4.3.6 Lessons Learned

This subsection outlines the most important lessons we have learned when adapting tasks for Duet.

Some tasks may require operating on a consistent view of at least a portion of the device or filesystem. For example, a file backup task may require that backed up files represent a consistent version of the data, unaffected by partial updates. Duet does not provide any such guarantees to tasks, apart from hints on data availability. The backup task we examined relies on the ability of Btrfs to take filesystem snapshots, to ensure backup consistency. Alternatively, a backup task could use file locking to ensure consistency at the file-level and leverage Duet events to prioritize files with in-memory pages, similar to our defragmentation task.

Tasks should not assume that data will be available (or unmodified) after being notified about an event. This helps avoid races and inconsistencies. For example, our backup task locks a page before checking its dirty status and whether it belongs to the snapshot, as described earlier in Section 4.3.2.

Maintenance tasks may consume CPU and memory resources while running, which could affect the performance of workloads. In our experience, maintenance tasks make moderate use of these resources, as they are usually bottlenecked on I/O. Thus, an I/O scheduler capable of assigning low priority to maintenance I/O works well [66, 84, 151]. Furthermore, maintenance work is usually partitioned in small chunks that can be scheduled around workloads. For example, rsync processes files in 32KB chunks. Overall, Duet is not dependent on the way that maintenance work is scheduled or partitioned, allowing tasks to individually regulate their impact on workloads.

4.4 Evaluation

This section evaluates the benefits of Duet. Our evaluation has three goals. First, we evaluate the ability of Duet to reduce I/O when a maintenance task runs together with a foreground workload. Second, we evaluate the I/O reduction when maintenance tasks are run concurrently, which implicitly enables them to collaborate on shared data. Third, we evaluate the overhead of Duet.

Section 4.4.1 describes our experimental setup. Section 4.4.2 and Section 4.4.3 quantify the I/O reduction when running maintenance tasks individually and concurrently. Section 4.4.4 evaluates the overhead of Duet using microbenchmarks. Section 4.4.5 concludes our evaluation by discussing the effect of other parameters on our approach.

4.4.1 Experimental Methodology

We have chosen to use the Filebench benchmark [77] as the foreground workload in our experiments. Filebench is a widely used benchmark that allows us the flexibility to change most aspects of its workload, allowing us to evaluate Duet for a range of workload characteristics.

An alternative would be to use real traces of file system activity for the workload, but we found few publicly available traces that can be replayed accurately [249]. Moreover, the traces did not contain
Chapter 4. Opportunistic storage maintenance

sufficient information needed for our evaluation. For instance, existing traces do not provide information on files that are not accessed (or the corresponding fraction of the file system), but we need this information for some maintenance tasks. Using existing traces also does not allow for adjusting any workload parameters.

Workload Characteristics

Three workload characteristics have the most impact on our opportunistic approach: data overlap, read-write ratio and workload I/O rate. Next, we describe how we vary these characteristics to study their effect in our experiments.

Our approach reduces the I/O footprint of a maintenance task when the data accessed by the task overlaps with the data accessed by other ongoing maintenance tasks or with foreground workloads. While there is potential for high data overlap between maintenance tasks running concurrently, the data overlap with workloads depends on both the type of maintenance task and the workload. Many tasks, such as incremental backups and garbage collectors, tend to access hot areas of the file system, so data overlap with the workload is expected to be high. For tasks that access all data on a device, such as scrubbing, the data overlap will vary depending on the workload since there is high variability in the fraction of device data accessed across workloads [34, 194].

By default, Filebench uses a uniform distribution to pick the files it operates on, which gives it high coverage of the file system, i.e. a large percentage of the files get accessed, creating high data overlap with maintenance work. We modified Filebench in two ways to vary the amount of data overlap. First, we limit the data coverage of Filebench to different fractions of the overall file system. Second, we analyzed the Microsoft Production Build Server trace [119] and extracted traces of file events for three different storage devices. Figure 4.1 shows that the file access distributions of the Microsoft traces are highly skewed compared to Filebench’s uniform distribution policy. We have modified Filebench to pick files using the Microsoft distributions, and we show results for both the uniform and the skewed file access distributions.

The read-write ratio of workload operations also impacts opportunistic processing because maintenance tasks react differently to updates. For example, a file may be further fragmented or defragmented due to a write, and sharing with the backup snapshot is broken when a block is updated. We use

Figure 4.1. File access distributions for Microsoft traces and the Filebench benchmark
Chapter 4. Opportunistic storage maintenance

<table>
<thead>
<tr>
<th>Metric</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>I/O saved</td>
<td>Maintenance I/O saved with Duet</td>
</tr>
<tr>
<td></td>
<td>Total maintenance I/O performed without Duet</td>
</tr>
<tr>
<td>Maximum utilization</td>
<td>Maximum device utilization by the foreground workload, at which</td>
</tr>
<tr>
<td></td>
<td>maintenance work completes by the end of the experiment</td>
</tr>
<tr>
<td>Speedup</td>
<td>Task completion time without Duet</td>
</tr>
<tr>
<td></td>
<td>Task completion time with Duet</td>
</tr>
</tbody>
</table>

Table 4.4. Duet evaluation metrics

Filebench with three of its default workload personalities. The fileserver personality is a write-heavy workload, with a read-write ratio of 1:2. The webproxy personality is more read-heavy, with read-write ratio of 4:1. Finally, the webserver personality is a read-mostly workload with a 10:1 read-write ratio, with all write operations appending data to a single log file. Note that maintenance operations are useful even on systems that run read-mostly workloads. For example, scrubbing can help detect and repair data corruption caused by hardware failure, and similarly, backup can help with recovery from software bugs, administrative errors or security incidents.

Finally, the workload I/O rate affects the opportunistic processing performed by Duet tasks. Increased workload I/O rate creates more opportunities for synergy with maintenance work, but it reduces the overall time that the storage device remains idle. Since maintenance tasks run at idle times to reduce their impact on workloads, they need enough idle time to complete their work. In our experiments, we control workload I/O through rate-limiting commands available in Filebench to throttle its bandwidth.

Evaluation Metrics

Our evaluation uses three metrics: I/O saved, maximum utilization, and speedup. The first metric measures the maintenance I/O saved by Duet. The second metric takes into account that when tasks only run at idle periods, they may not complete under high device utilization. We define device utilization as the percentage of time during which foreground I/O requests keep the device busy, when no maintenance tasks are being run. This metric is reported as the \%util statistic of the iostat tool. We profiled each Filebench personality with different levels of throttling (and no maintenance load) to achieve a given device utilization, and report results for utilization values ranging from 0-100%, in 10% intervals. The maximum utilization is the highest device utilization at which maintenance work can still be completed in 30 minutes, by the end of the experiment. Finally, when tasks run with normal IO priority, such as Rsync, we run Filebench unthrottled, and measure the speedup of the maintenance task. Table 4.4 summarizes these metrics; higher values are always better.

Experimental Setup

We conduct our experiments for 30 minutes on a file system populated with 50GB of data, during which maintenance tasks run concurrently with the workload. At this rate, maintenance can be run weekly on a 16TB storage system. Each experiment is run three times, and every data point in our plots is
an average across these runs. Generally, there is low variability across the runs, and so we omit the error margins. Otherwise, we show 95% confidence intervals. We use CFQ, the default Linux I/O scheduler that supports I/O prioritization. Our in-kernel tasks issue their maintenance I/O requests at Idle priority. These requests are serviced only after the device has remained idle for some time. We have measured the latency of Filebench workloads, both without and with one or two maintenance tasks running concurrently, at various device utilizations, and found that there is insignificant impact on workload latency, as shown in Figure 4.2. As an example, the webserver workload latency at 50% device utilization, without any maintenance task is $11.67 \pm 0.12 \text{ms}$. When scrubbing, it is $11.60 \pm 0.25 \text{ms}$, and with backup it is $11.82 \pm 0.16 \text{ms}$.

All experiments are run on HP ProLiant DL160 Gen8 servers, equipped with Intel Xeon E5-2650 CPUs with 8 cores, and 300GB SAS drives running at 10K RPM. While the machine has 32GB of DDR3 RAM, we boot it with 2GB of memory to have a realistic page cache size compared to our working set of 50GB. We examine the effect of the page cache size on our approach in more detail in Section 4.4.5.
Chapter 4. Opportunistic storage maintenance

4.4.2 Running Single Tasks

This section evaluates the ability of Duet to perform maintenance work opportunistically. We evaluate Duet with five different maintenance tasks, while varying the data overlap between the maintenance and foreground work.

Scrubbing. We implemented opportunistic scrubbing by modifying 75 of the 3500 lines of code of the Btrfs scrubber. Our evaluation with different Filebench workloads shows that, as expected, the I/O saved with opportunistic scrubbing increases with higher device utilization and the data overlap between the workload and the scrubber. Figure 4.3 shows the results for the webserver workload. As utilization and data overlap increase, more data gets accessed by the workload, avoiding the need to scrub it. Beyond a certain utilization, the I/O saved reaches an upper limit, equal to the data overlap. At this point, the scrubber skips scrubbing all shared data because the workload accesses it before the scrubber processes it as part of its sequential scan, showing that Duet allows exploiting any available synergy between I/O tasks.

Savings decrease for more write-heavy workloads. Recall that we unset the done bit for updated blocks, if they have not already been scrubbed in the course of normal (sequential) scrubbing. The webproxy performs similarly to the webserver because its write operations mainly append data to files, allowing read operations to significantly reduce scrubbing work. However, the write-intensive fileserver workload has 40% of the IO savings compared to the other two workloads, since any file can be overwritten, so the opportunistic savings are lower. When the skewed file access distribution is used, the results are similar, but savings are decreased by 15-30%. This decrease is small, despite the majority of accesses being directed to a small fraction of the files, because it is sufficient for a file to be read once to be considered scrubbed.

By reducing the required I/O, Duet allows scrubbing to complete faster. Table 4.5 shows the max-

Table 4.5. Maximum utilization with and without Duet for Btrfs maintenance tasks: The table shows the maximum utilization (in 10% intervals) at which each maintenance task can still complete its work in a 30-minute interval. Higher values indicate that the foreground workload is able to utilize the device more. Higher read-write ratios, higher data overlap and a uniform distribution of file accesses improve opportunistic processing, allowing higher maximum utilization.
imun device utilization at which scrubbing completes in a 30 minute interval. Note that for normal scrubbing to complete, the device must not be busier than 70% (column 4), regardless of the workload, because the amount of work remains constant. With Duet, the maximum utilization increases with the data overlap. Devices can be busier, from 70% to 100% (column 5), depending on the characteristics of the workload.

**Backup.** We implemented opportunistic backup by modifying 140 of the 4900 lines of code of the Btrfs backup tool. The backup tool processes files in the order of their inode numbers, and each file is processed fully before moving to the next one. This results in more random access than scrubbing, and so the backup requires almost twice the amount of time needed for scrubbing. This extra time allows the backup task to interact longer with the foreground workload, resulting in more opportunities for I/O savings. Therefore, the I/O saved reaches its upper limit at a much lower device utilization compared to scrubbing. For example, Figure 4.4 shows that with 25% overlap, the maximum I/O saved is reached at 20% utilization versus 40% for scrubbing (see Figure 4.3).

With the additional seeks, the baseline can tolerate a maximum utilization of 40%, which is close to half that for baseline scrubbing (columns 6 and 4 in Table 4.5). Duet reduces random I/O, allowing backup to complete on devices with 50-100% utilization (25-150% busier than the baseline).

When a block is updated, it gets copied to a new location and is no longer shared with the backup snapshot. Therefore, subsequent reads to the same file offset do not benefit the backup task, as they no longer refer to the snapshot data. As a result, the I/O saved decreases with decreasing read-write ratio. This effect applies both to writes that append and overwrite data, as well as to deletions and recreations of files. Webproxy, which includes file append, delete, and create operations shows the impact of breaking sharing with the backup snapshot in this way. It yields 80% of the I/O savings of webserver, while fileserver, which also breaks sharing by overwriting files, yields up to 40% of the IO savings of webserver.
Defragmentation. We implemented opportunistic defragmentation by modifying 95 of the 1200 lines of code of the Btrfs defragmenter. The defragmenter merges small, logically adjacent extents by bringing them into memory, and then writing them back to storage as part of the same transaction, thus creating a single, larger extent. The total I/O required to defragment a file consists of the number of pages read and then written, which is twice the number of pages in the new extent. Recall that we reduce this I/O by prioritizing files that have more pages in memory. Therefore, the I/O saved is the sum of the number of pages in memory when an extent is processed and the number of pages that were already marked dirty by the workload. The former do not require I/O and the latter will be flushed soon anyway. Our experiments are performed on a 10% fragmented file system.

As shown in Figure 4.5, the I/O saved results with defragmentation are similar but smaller than the scrubbing and backup results shown in Figures 4.3 and 4.4. For read-heavy workloads such as webserver, we only save on read accesses, which are close to 50% of the total I/O. On the other hand, a workload appending data to files, such as webproxy, can also save I/O by defragmenting files with dirty pages in memory. This benefit does not apply to a write-heavy workload like fileserver, which overwrites files thus defragmenting them, reducing the total work performed by the maintenance task. In this case, the savings are available for only those files that the opportunistic defragmentation task processed. Similar to scrubbing and backup, using the skewed file access distribution reduces the I/O saved by 15-30%.

The maximum utilization results for defragmentation are similar to the previous two tasks, and shown in the last two columns of Table 4.5. The difference between the baseline and Duet is less pronounced compared to the other workloads, because most defragmentation writes still need to be performed. As a result, we improve upon the baseline by at most 50% across different workloads.

Garbage collection. The opportunistic garbage collector was implemented by adding 150 lines to the 1400 lines of code of the F2fs in-kernel garbage collector. Our aim is to clean segments faster by selecting segments with cached blocks. Reducing the segment cleaning time is crucial when the file system is running out of clean segments. In that case, F2fs transitions to overwriting invalid blocks in scattered segments. When that happens, we have measured a 57% increase in filebench latency, and

---

2The page could be modified again between the time we flush it as part of the defragmentation process, and the time when it was originally planned to be flushed. We cannot account for this case.
Table 4.6. Segment cleaning time with and without Duet

<table>
<thead>
<tr>
<th>Workload, device utilization</th>
<th>Segment cleaning time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Baseline</td>
</tr>
<tr>
<td>Fileserver, 40%</td>
<td>17.1 ± 3.5ms</td>
</tr>
<tr>
<td>Fileserver, 50%</td>
<td>17.0 ± 1.1ms</td>
</tr>
<tr>
<td>Fileserver, 60%</td>
<td>16.4 ± 1.4ms</td>
</tr>
<tr>
<td>Fileserver, 70%</td>
<td>15.8 ± 1.0ms</td>
</tr>
</tbody>
</table>

Figure 4.6. Runtime speedup with different overlap for Rsync

29% increase in device utilization. However, even when there is no pressure for clean segments, speeding up cleaning time enables consuming less idle time or cleaning more segments.

We have used the fileserver workload for these experiments because it is the only workload that overwrites and deletes existing blocks. We present our results when the fileserver is run between 40% and 70% device utilization. At lower utilization, the garbage collector does not run, and at higher utilization, there is not enough idle time for the garbage collector to run. Table 4.6 shows the average cleaning time for a segment, with and without Duet. Duet improves cleaning performance at higher utilization, when cleaning is most needed. Performance improves because more segment blocks are cached in memory, and the opportunistic garbage collector picks segments requiring fewer read operations.

Rsync. We implemented opportunistic Rsync by modifying 300 of the 45000 lines of code in the Rsync application. We evaluated opportunistic Rsync by running it locally, copying 50GB of data between two disks, while running Filebench on the source device during the transfer. Rsync is used locally for various tasks, such as when performing snapshot-based backups [238], synchronizing data across VMs, and for copying data when upgrading devices [236].

The total I/O required to synchronize a file between the source and destination folders includes reading all of its data at both the sender and the receiver side to produce checksums, and writing the updated data blocks on the receiving side. In our experiments, the destination folder is initially empty, so the files are not checksummed. Instead, their data is sent to the receiving side and the I/O operations required per file are twice the number of data blocks of the file, for reading and writing each once.
Workload, Read-Write ratio | Overlap | File accesses | Scrubbing and backup | Scrubbing, backup, defragmentation
--- | --- | --- | --- | ---
Webserver, Ratio 10:1 | 25% | Uniform | 30% | 70% | – | 30%
| 50% | Uniform | 30% | 80% | – | 40%
| 75% | Uniform | 30% | 90% | – | 40%
| 100% | Uniform | 30% | 90% | – | 50%
| 100% | MS trace | 30% | 90% | – | 20%
Webproxy, Ratio 4:1 | 100% | Uniform | 30% | 90% | – | 20%
| 100% | MS trace | 30% | 70% | – | 20%
Fileserv, Ratio 1:2 | 100% | Uniform | 30% | 80% | – | 30%
| 100% | MS trace | 30% | 70% | – | 20%

Table 4.7. Maximum utilization with and without Duet when multiple Btrfs tasks are combined. Similar to Table 4.5, we show the maximum utilization (in 10% intervals) at which each combination of maintenance tasks can still complete their work in a 30-minute interval. Higher values indicate that the foreground workload is able to utilize the device more. We find that without Duet, even low-utilized devices can struggle to complete all maintenance work.

Similar to our previous experiments, we find that read I/O can be reduced proportional to the data overlap between the workload and Rsync. Similar to defragmentation, write I/O cannot be saved, so with 100% overlap we can save 50% of the total I/O.

In the previous experiments, the maintenance task is run at a lower priority, and thus the Filebench workload is run throttled to allow the maintenance task to make progress. Rsync, however, runs at normal I/O priority, affecting Filebench throughput by up to 27%. Thus, in this experiment we run Filebench unthrottled and measure the speedup of Rsync. Figure 4.6 shows the results for the webserver workload. It shows that the speedup increases with higher data overlap, with Rsync completing twice as fast at 100% data overlap. This speedup reduces the period during which Rsync impacts the workload.

4.4.3 Running Multiple Tasks Together

Today, maintenance tasks are run in isolation to avoid interference and slowdown. This section shows that when Duet tasks are run concurrently, I/O savings increase due to higher data overlap, enabling them to complete their work faster.

Scrubbing and Backup. In this experiment, we run scrubbing and backup together with the different Filebench workloads. Figure 4.7 shows the results for the amount of I/O saved for the webserver workload. With Duet, data accesses by either the backup task, or the scrubber, benefit the other task. As a result, even when Filebench is not run (0% utilization), Duet reduces the total I/O needed to complete maintenance work by at least 50%. Similar to previous results, higher device utilization and higher data overlap increase I/O savings further. The results for other workloads are similar to the results discussed previously in Section 4.4.2, with more write-intensive workloads resulting in lower savings.

The significant work reduction allows us to also complete maintenance work on busier devices. Figure 4.8 shows the maintenance work completed at various device utilizations. While the baseline tasks fail
Chapter 4. Opportunistic storage maintenance

Figure 4.7. I/O saved when the scrubbing and backup tasks are run together with the webserver workload

Figure 4.8. Maintenance work completed when scrubbing and backup are run together with the webserver workload

Figure 4.9. I/O saved when scrubbing, backup, and defragmentation are run together with the webserver workload
to complete maintenance work beyond 30% device utilization, Duet allows 70-90% maximum utilization. Results for different workloads are shown in Table 4.7.

**Scrubbing, Backup, and Defragmentation.** We also experimented with combining three maintenance tasks. As shown in Figure 4.9, roughly 55% of maintenance I/O is needed when the tasks run without Filebench, as all three are accessing the same data. This I/O includes one pass over the file system, and the write requests for the defragmenter, which cannot be saved. The maximum I/O saved with the read-only webserver workload is roughly 80%, which is almost all maintenance work, other than the writes needed for defragmentation. More write-intensive workloads perform worse, but still achieve I/O savings up to 60%.

Figure 4.10 shows the maintenance work completed at different device utilizations. Duet completes all maintenance work even with 50% device utilization, which the baseline can complete only 25% of the work even on an idle device. All results for the webserver workload, as well as other workloads we have evaluated, are summarized in Table 4.7.

### 4.4.4 Performance Overhead

**CPU overhead** To determine the overhead of Duet, we run a simple file task that registers the root directory of the file system with Duet, and either remains idle, or fetches events periodically in 10, 20 and 40ms intervals, sleeping in between. We chose these intervals because they are close to the typical Rsync fetch interval, which is 20ms. To generate page events, we run the webserver workload unthrottled on the file system, which generates roughly 12 page events/ms. We estimate the CPU available to applications by running a program that spins in a tight loop at low priority, and then measure the loop counter value periodically. Based on the counter value, the CPU overhead of using Duet is roughly 0.5-1.5%, as shown in Figure 4.11. State-based notifications have slightly lower overhead because events can be merged.
Chapter 4. Opportunistic storage maintenance

![Graph showing CPU overhead of Duet](image1)

**Figure 4.11.** CPU overhead of Duet

![Graph showing I/O saved on a solid-state drive](image2)

**Figure 4.12.** I/O saved on a solid-state drive

The fetch frequency does not change the number of events that need to be copied to user space and thus has a small effect on overhead.

**Memory overhead** Duet maintains item descriptors for pages and bitmaps for up to $N$ concurrent sessions. For $N = 16$, an item descriptor requires 32 bytes (inode number, offset, 16-byte flag array and hash node). With state notifications, the worst case memory overhead is 1.5% ($\frac{32 \times 2}{4096}$), as explained in Section 4.2.2. In practice, fetch is called often enough that a buffer of 256 items does not fill up. At most, such a buffer would require 2.3KB, since we only return the flag variable for one session, and no hash node.

Item bitmaps are dynamically allocated when the range that the bitmap represents contains both set and unset bits. In the worst case, block tasks will use 1 bit per device block, and file tasks will use 2 bits per inode. In our experiments, when scrubbing a fully utilized disk with 100% overlap with the workload, the bitmap required 1.47MB, while the worst case estimate for 50GB of data is 1.56MB.
4.4.5 Effect of Other Parameters

Storage device type. The results presented so far have been based on a hard drive setup. To evaluate the efficiency of Duet on faster devices, we experimented with a consumer-grade solid-state drive (Intel SSD 510 120GB), and found that for this model, the I/O savings and the work completed did not change qualitatively. For example, the scrubber completes in half the time, but the throughput of the workload is also much higher, resulting in the same number of accesses and savings, as shown in Figure 4.12. With backup, which exhibits an I/O pattern that consists of more random 64KB requests, we achieve higher savings on the SSD. The reason is that the random read performance of our Intel 510 SSD and our enterprise 10K RPM hard drive is roughly similar for 64KB requests, at about 21 MB/s [92]. Hence, the default backup time is similar on the hard drive and the SSD. Since the Filebench workload is more sequential, it has higher throughput on the SSD and ends up directing the order that the backup processes data. This allows more data overlap and thus the Duet-enabled backup achieves higher I/O savings.

I/O prioritization. Our in-kernel maintenance tasks were run at lower priority, which has minimal impact on the workload. We also experimented with the Linux Deadline I/O scheduler, which does not allow prioritizing different streams of I/O. We find that without I/O prioritization, workload requests are slowed down significantly when a maintenance task is running. Maintenance work finishes faster but the workload issues fewer data requests and thus the I/O saved is reduced. Hence, Duet works better when maintenance tasks run at low priority.

Cold data placement. We define cold data as the part of the file system that is not accessed by the workload but requires maintenance. We find that the physical placement of this data on the storage device does not affect performance, even when cold data is separated from the data accessed by the workload. Since maintenance I/O occurs when the device has been idle, additional seeks occur only when switching between maintenance tasks and workloads.

Page cache size. We also modified the ratio of the page cache size to the file system size. This ratio is expected to affect the workload’s performance, but it may also affect the maintenance task. With a larger cache, more synergies and thus more I/O saving are expected. In our experiments, the page cache size is roughly 2% of the file system data accessed by the task. We find that cache size ratios in the range of 1-5% have a marginal effect on our results. We believe that maintenance tasks access significant amounts of data, and hence it is the out-of-order processing, rather than cache locality that provides the most benefits.

Order of maintenance initiation. Section 4.4.3 explored running maintenance tasks concurrently. Although each task is started at virtually the same time, the framework will store tasks in the order they were registered. When an event occurs, tasks will be informed in this same order. We experimented with changing the task registration order, and found that it had no effect on our results, because eventually, all tasks receive the data.
4.5 Extensions of the Opportunistic Work Model

We are currently applying the opportunistic work model, implemented through Duet, to contexts outside storage maintenance. Our initial choice of targeting storage maintenance tasks was attributed to their lack of hard deadlines, especially with regards to individual pieces of work, which allows us to freely reorder work without affecting guarantees. However, we find other applications can also benefit from information about the operating system’s cache contents. In the following sections, we describe our experience applying our opportunistic approach in two contexts: distributed computational frameworks (Section 4.5.1), and as an enhancement for applications that rely on file notification frameworks (Section 4.5.2). We further provide encouraging preliminary results suggesting that our opportunistic model is a good fit in these environments.

4.5.1 Opportunistic computational frameworks

Cluster computing frameworks such as Apache Hadoop [16] and Apache Spark [17] are commonly used to run a variety of data analytics applications, helping uncover correlations and trends in large data sets. These frameworks allow users to focus on their particular data analysis problem, while handling the complexities of distribution, including data placement and replication, computation placement, fault tolerance, and resource negotiation in a shared cluster. The users of these frameworks span many disciplines and business types. Web search companies collect and analyze vast web topology data [86, 253], storage and security companies collect customer telemetry data to study how their software products behave in the field [9, 252, 254]; social networking companies store and analyze data on user interactions [132, 149]; businesses rely on data warehouses to produce periodic reports on business performance. On the academic side, computer scientists collect terabytes of network traffic data to improve the efficiency of routing algorithms [244]; and biologists collect genome sequencing data to discover connections between genes and gene products [160], among many other uses. In all these scenarios, historical data is typically retained indefinitely due to its value.

Unfortunately, the exponential growth in the volume of data collected and used for analysis [208] has not been matched by corresponding increases in hard disk performance, where this data is primarily stored [42]. As a result, performing analysis on even the most recent part of a dataset may be limited by disk access speeds. Furthermore, a common analysis pattern involves running multiple, similar queries over the same data. For example, during typical exploratory analysis of a dataset, a data scientist may dispatch multiple queries that differ marginally from each other. Evidence of this work pattern is seen in a recent study of three academic Hadoop clusters by Ren et al. [183], which found that less than 10% of input files are responsible for more than 80% of all accesses, and that only 1% of datasets are shared across users. They further found that 90% of data re-accesses happen within 1 hour, and that 35-60% of job pipelines\(^3\) are submitted within 10 seconds after an earlier pipeline from the same user finishes executing. In other words, users commonly submit a series of jobs that access the same input datasets in a relatively short period of time. Similarly, in production settings, multiple data scientists may run independent analyses that access a large portion of a common dataset concurrently.

Users of cluster computing frameworks enjoy limited benefit from this data sharing today. MapReduce, for example, only considers disk and rack locality when making task placement decisions. It does not take memory locality into account, possibly because it assumes that massive data sets will

\(^{3}\)A pipeline is defined as a sequence of jobs where the output of one job forms the input of the next job in the pipeline.
be displaced from memory before they can be reused. The more recent caching support added to the Hadoop Distributed File System (HDFS) requires users to explicitly identify files that should be pinned in memory [18]; these pinned files are then also considered during task placement. This explicit cache management, however, both increases the burden on users and is ineffective when the reuse occurs across multiple jobs operating on datasets that are larger than available memory. Users similarly expect caching to matter little and tend to run jobs serially even when they are not dependent on the results of the previous jobs [183].

Another challenge for exploiting data reuse in current frameworks is that the order in which jobs process data is agnostic to the data availability in the cache, which can lead to unnecessary cache evictions and disk accesses. As a result, even when the inputs of two jobs overlap completely, if they are scheduled even slightly apart in time they can miss opportunities for sharing data accesses. In our cluster, scheduling two identical Spark jobs 4 minutes apart increases their runtime by 145% compared to starting them together.

We believe that jobs on frameworks such as Hadoop and Spark should be able to benefit from data reuse when jobs share any portion of their input. To achieve that, we are working on an approach that leverages the fact that tasks within a job are independent of each other, which is a core requirement in these frameworks for scalability and for task re-execution to mask failures and stragglers. Our insight is that we can reorder tasks within a job to prioritize the processing of data brought into memory by other jobs, without affecting correctness. To inform tasks of data available in memory we use Duet to expose page cache information [9]. Using this information, we are working on harmonizing task scheduling with caching in these frameworks.

**Harmonizing task scheduling**

To enable users of distributed computational frameworks to benefit from inter-job data sharing, we performed a set of changes to the architecture of existing frameworks. Our changes, which are currently available for both Hadoop and Spark, are concentrated on the task scheduling component, which is typically implemented in a modular fashion in these frameworks. As a result, while Hadoop and Spark are different frameworks, written in different languages, their internal structures were similar enough to allow these changes to be implemented in less then 500 lines of code each. This enforces our belief that our modifications could be applied to other systems with similar effort.

We collectively refer to our architectural changes as Quartet. For both Hadoop and Spark, our current implementation of Quartet consists of four components. First, the *Quartet Watcher* is a service running on each node of the cluster, using *Duet* to track changes in the operating system cache. These events are aggregated and periodically reported to the third component of the system, the *Quartet Manager*. Specifically, the Manager periodically collects reports from the Watchers that state what portion of individual HDFS blocks are contained in each cache. Then, the *Application Master* component registers blocks of interest for individual jobs with the Manager. The Application Master is responsible for scheduling job tasks, so we have modified it to account for cached blocks when making scheduling decisions. Specifically, using information on the cache residency of blocks of interest received from the Manager, we prioritize those tasks within a job that have most data in the cache of a node in our cluster.
Evaluation

Our preliminary results on Quartet are encouraging. As a first step, we have evaluated Quartet by measuring the cache hit rate and runtime of jobs that are scheduled after jobs operating on the same dataset. Our experimental setup consists of a cluster of 24 worker nodes, each configured with an 8-core Intel Xeon L5420 CPU, 16 GB RAM and a 1 TB 7200 RPM hard drive. In order to use Duet, these nodes ran a modified Linux 3.13.6 kernel. A separate, identical node was dedicated to running the HDFS Name Node and Quartet Manager services, in addition to the YARN Resource Manager for Hadoop (resp. the Spark Master for Spark). We configured HDFS with three replicas and 128 MB blocks. Each worker node was allocated 8 concurrent YARN containers (resp. Spark Executors). The vanilla Hadoop version was 2.7.1, and vanilla Spark was 1.6.0. The Quartet modifications were made to each of these versions.

We measure the effectiveness of Quartet through an experiment based on a real-world workload. Ren et al. report that 90% of data re-accesses occur within one hour of the last access on two different Hadoop clusters [183]. To approximate this scenario, we run two jobs that access the same dataset sequentially. We repeated our experiment using three files of different sizes: 256 GB, 512 GB, and 1 TB. The first of these fits completely within the physical memory of the cluster (384 GB) while the latter two exceed it. For our experiments, we used a custom line counting application for both Spark and Hadoop, to simulate an I/O-bound workload.

**Disk accesses.** Figure 4.13 shows the percentage of data accesses that were satisfied from the cache, for the second of two identical jobs. We show results for both Hadoop and Spark, with and without Quartet. As expected, for 256 GB jobs most of the blocks are still in the page cache when the first job finishes. Quartet can take advantage of that fully, demonstrating cache hit rates of 92-98%. In the case of vanilla Hadoop and Spark, however, cache hit rates are as low as 42-44%. We attribute these reduced cache hit rates to HDFS storing multiple replicas of every data block, and the way these replicas are selected by the scheduler. HDFS in our cluster uses the default configuration of three replicas per block. Although all blocks are cached across the cluster when the first job finishes, the second job’s tasks are often assigned different replicas of the same blocks by the scheduler. For job inputs larger than the cluster memory, such as 512 GB and 1 TB, part of the input data was already evicted from the cache by the end of the first job. This, coupled with replica selection in HDFS results in less than 4% cache hit
rates. When Quartet is enabled, however, resident blocks are prioritized, and cache hits rates of 25-56% are possible.

**Runtime.** Reducing the amount of I/O for I/O-bound jobs is also expected to reduce their runtime. Overall, we find that Quartet improves most on Spark job runtimes, because it reuses Executors, while Hadoop launches new JVM containers for each task. As a result, the cost of setup and teardown of containers reduces the effect of Quartet’s runtime improvements. More specifically, when job data fits entirely in memory the runtime of the second job can be reduced by an additional 45% for Spark and 2.7% for Hadoop using Quartet, compared to the vanilla versions of both frameworks. When the job data exceed our memory capacity, Quartet on Spark improves on job runtime by an additional 21-43%, while Quartet on Hadoop improves runtimes by 6-13%.

**Overhead.** On each of the worker nodes, our prototype Watcher implementation adds less than 20% CPU overhead on a single core, while the Manager itself consumed less than 5% CPU usage on a single core. The network traffic between the watcher, the manager, and the applications is proportional to the number of HDFS blocks with page cache updates, and the update rate requested by the application. In our experiments updates are requested once per second, and this traffic is in the order of 10-100 KB/s per application.

### 4.5.2 Complementing file notification frameworks

A large number of applications today need to be reactive to changes in data. Maintenance tasks that have been extensively covered in this thesis, such as incremental backup jobs, defragmentation processes, and anti-virus scanners, often need to operate on modified data. Applications that are costly to restart, such as web servers and databases, need to become aware of changes in configuration or critical files. System utilities, such as file managers, search indexes, and garbage collectors, can provide better services by tracking data changes. Due to their prohibitive cost and their impact on the system, full file system or device scans are generally avoided in these scenarios. Instead, a common approach is polling individual files to receive notifications on updates. However, the polling approach tends to be inefficient and hard to scale, and may miss certain types of changes, such as metadata updates. Another issue with this approach is the timeliness of the updates. Data integrity systems like Tripwire [124] track file changes based on a fixed schedule, but this approach does not work for applications that require real time updates, such as Dropbox [65], Box [39], Microsoft OneDrive [153], Google Drive [85], and open-source solutions such as Seafile [203] and ownCloud [166].

File notification frameworks fulfill these requirements: Inotify [105, 122] in Linux, kqueue [133] in BSD, FSEvents [19] in OS X, and the FileSystemWatcher .NET class [155] in Windows provide an efficient way to trace actions taking place at the file system in real time. In the case of Inotify, changes are tracked through hooks in the virtual file system layer, and events are stored in a queue of predefined size. Events are accessed by applications through a file descriptor created at registration time. To receive events on a given directory, applications are required to explicitly add a “watcher” on it through an API call, informing the framework of their interest. Events are recorded only after the watcher has been placed.

Unfortunately, notification frameworks today are limited to file-level events. As a result, applications are informed of data accesses on a file, i.e., read and write operations, but they need to implement their own mechanisms to discover the offset where the access occurred. While this is possible for writes, through the comparison of checksums before and after the change, it becomes impossible for reads. Other
shortcomings of the Inotify framework include the absence of support for recursive directory watches, meaning that a separate Inotify watch must be created for every sub-directory, and indirect support for rename operations, which are handled as two separate events (file removal and subsequent file insertion to the directory) and must be examined and matched in a context of potential race conditions.

Duet can complement the operation of these frameworks by providing page-level events to applications. This way, no additional logic is required to infer the offset of data accesses within files. Furthermore, we have described that Duet already has built-in support for recursive directory watching. This reduces the effort required by applications, which consists of full scans of watched directory tree. Finally, due to the nature of file-level events, it is hard to bound the queue size of the queue holding them to ensure that events will not be dropped. On the contrary, Duet can guarantee that page-level access events will not be dropped, if it is allocated a predefined amount of memory that is dependent on the page cache size (see Section 4.2.2). We are currently working on adapting applications to use Duet events as complementary to Inotify events, which can significantly simplify their event-handling logic and improve their performance.

Application to file synchronization frameworks

We have applied our opportunistic work model to a file synchronization application, Seafile [203]. Similar to other applications of the same type, Seafile stores files on a central server, and allows personal computers and mobile devices to synchronize their local data via the Seafile client. Seafile’s functionality is similar to other popular services such as Dropbox and Google Drive, but we have chosen to work with Seafile because it is open source. As a result, a primary difference between Seafile and proprietary solutions is that it enables users to host their own Seafile servers. Similar to Dropbox, Seafile’s Linux implementation uses the Inotify framework for file notifications.

Seafile operates using a backend based on the architecture of the popular Git versioning system. Specifically, each client views their local data as a branch of the repository hosted on the server. Every time a local modification is detected, the new data is committed locally. A periodic synchronization process consists of a merge operation of the local branch with the server’s master branch. Unlike Git, however, conflicts are resolved by renaming the conflicting file versions. In order to detect file modifications online, Seafile relies on Inotify notifications. Since Inotify does not provide information on the file offset affected by the modification, additional processing logic is required. To detect the modified offset, Seafile relies on content-defined chunking based on the principles of the Low-Bandwidth Network File System [158]. Specifically, files are broken into variable-sized chunks, with chunk boundaries defined using the Rabin fingerprinting algorithm [179]. Each chunk is internally represented by its offset, and a SHA-1 hash. In order to detect a modification offset, Seafile needs to scan the affected file in its entirety, comparing chunk offsets and hashes.

We find that the amount of time spent to complete the file chunking phase makes up a significant portion of Seafile’s synchronization process. Our experimental setup consists of two HP ProLiant DL160 Gen8 servers, equipped with Intel Xeon E5-2650 CPUs with 8 cores, 300GB SAS drives running at 10K RPM, and 32GB of DDR3 RAM. One of the servers is configured as the Seafile server, and the other as the Seafile client. We generate files with random contents, modify each at a uniformly randomly selected offset, and then trigger the Seafile synchronization process. We have run multiple iterations of different configurations with Seafile repositories up to 15GB in size. In Figure 4.14, we show both the total synchronization time (labelled as Synchronization), as well as the chunking time (labelled as...
Chapter 4. Opportunistic storage maintenance

Figure 4.14. Total time taken to synchronize and chunk Seafile repositories of different sizes with, and without using Duet events.

Chunking) as a function of the repository size. It is evident that chunking makes up the majority of the synchronization process time, up to 92.6% on average.

With the help of Duet events, Seafile can be informed of the exact offset of file modifications. We have modified Seafile, simplifying its chunking code to rely on Duet Modified events instead. Using this additional information, Seafile can omit scanning and chunking the file data prior to the modified offset. This simple approach, which we use as a proof of concept, does not consider omitting unmodified data at the end of the file. We are currently working on a more sophisticated algorithm. We repeated our experiment described above with, and without the use of Duet events. Figure 4.14 shows the amount of time spent by Seafile synchronizing files and performing chunking operations with Duet, labelled as Synchronization with Duet and Chunking with Duet respectively, as a function of the repository size.

We find our preliminary results encouraging, with Duet reducing chunking time for Seafile by 38.2% on average across repository sizes, and by as much as 55.0% for repositories larger than the available 32GB of RAM on our machines, which require more I/O to be synchronized. Finally, we find that the total synchronization time decreases proportionally to the improvement in chunking time for repositories up to 15GB. For larger repositories the synchronization time improves at a slower rate, which is likely a result of Seafile’s protocol performing many operations of the synchronization process in a synchronous fashion. We are looking into relaxing these properties to improve synchronization time further.
Chapter 5

Related work

This chapter provides a review of the literature that is relevant to the work presented in this thesis. In Section 5.1, we outline the different types of maintenance tasks that are used in modern storage systems to provide a host of guarantees. These guarantees can cover a wide spectrum: increasing reliability, improving long-term storage performance, enhancing security, etc. Although our list is not comprehensive of all tasks used in commercial systems, we describe some of the most popular ones and outline the reasons that make them a good fit for our work. In Section 5.2 we examine the existing literature on backup systems in detail, and its relation to our studies.

As we have discussed in Chapter 2, maintenance tasks need to be run periodically in order to be effective. Prior to our work, there have been many attempts to schedule I/O at times when it does not impact user applications. We review this work in Section 5.3. At a high level, the two most popular approaches include careful I/O request scheduling that exploits device characteristics to perform work for “free”, and scheduling work at times when the device is otherwise idle. We look into related work on both approaches in Sections 5.3.1 and 5.3.2.

An approach we discuss in Chapter 4 is enabling tasks to collaborate, by opportunistically processing data that is available in the cache. There is a rich literature that examines how to expose different types of local information to improve the overall efficiency of the system. Related work includes approaches that target database query processing engines, data caching across distributed systems, and information-centric networks, to name a few. In Section 5.4 we examine the literature that is relevant to opportunistic data processing in more detail.

5.1 Types of maintenance tasks

As cloud storage solutions are adopted by corporations and individuals, remote machines are entrusted with a consistently increasing volume of data of varying importance, from personal communications to internal business data. At this scale, data loss events and performance degradation can severely affect large groups of people or corporations. Specifically, it has been shown that the vast majority of information today is stored in hard disks [97], and disk drives can fail in numerous ways [28, 29, 199, 201]. Hence, storage reliability mechanisms are of paramount importance today more than ever. In addition to that, administrators are taking the opportunity of centralized storage to reduce storage costs by storing data efficiently in bulk, but are also expected to guarantee its integrity and access latency. This has
resulted in modern systems deploying a plethora of maintenance operations, the most popular of which are described in this section.

5.1.1 Maintaining a reliable storage system

Hard disk drives have been shown to fail in numerous ways. Whole-disk failures, when the disk ceases to function upon failure, have been studied extensively. Gray and Ingen [90] reported error rates ten times higher than the rates expected from the disk vendor specification sheets. This result has been verified and further quantified in studies that span significantly larger scales, namely those by Pinheiro et al. [174], and Schroeder and Gibson [201]. Fortunately, protecting against whole-disk failures is not overly challenging due to a variety of proposed solutions based on component redundancy [112], or data redundancy and erasure codes [171]. The latter come in a variety of flavors of different complexity, all of which support recovery from at least two whole-disk failures [8, 36, 52].

Unfortunately, modern storage systems need to be shielded from a richer variety of errors. Bairavasundaram et al. [28] study the occurrence of latent sector errors, where a disk sector becomes inaccessible. Latent sector errors are found to exhibit significant temporal and spatial locality, for a non-negligible number of disks that are affected. Other types of sector errors include errors at the inter-sector level, also referred to as silent corruption. Such errors include torn/lost writes, where a write to multiple sectors results in only a few or no sectors getting updated, and misdirected writes, where some writes are redirected to the wrong sectors, but committed successfully [127]. Inter-sector errors such as checksum mismatches and parity inconsistencies have also been studied by Bairavasundaram et al. [29], with similar conclusions to those listed for latent sector errors, but with the addition of checksum mismatch events being correlated even across disks. To enable recovery from inter-sector errors, erasure codes are employed in intra-disk redundancy schemes [176]. In the case of intra disk redundancy, data is collected into reliability blocks, and for every group of \( k \) blocks, \( m \) parity blocks are calculated using an erasure code. Then, if \( k \) out of the \( n = k + m \) blocks can be accessed, data can be recovered.

Several studies have attempted to frame the factors that contribute to both disk failures [68, 70, 205, 206], and reliable failure predictors have been built using data collected from the drive’s firmware [145, 147]. Specifically, Ma et al. [145] use Naive Bayes to estimate the failure probability of individual hard disks and RAID arrays, using the number of reallocated sectors reported from the S.M.A.R.T. reports produced by the drive’s firmware. Manousakis et al. [147] present a model that extends the Arrhenius failure rate equation to account for relative humidity, the accuracy of which is also measured based on S.M.A.R.T. counters. These models are not developed as a replacement for periodic maintenance, however, for three reasons. First, they rely on support from the drive’s firmware, which is not always present or sufficient [103, 233]. Specifically, an extensive study by Google showed that half of the failed drives in their fleet never exhibits any reallocated sector errors according to S.M.A.R.T. reports [174]. Second, for the drives that do provide the proper firmware support, the accuracy of S.M.A.R.T. metrics is dependent on the workload, as they are updated when sectors get accessed. Therefore, it is suggested that the data be periodically accessed by a maintenance task, such as a scrubber [145]. Finally, our study presented in Section 2.1 shows that different failure rates occur across hard drive models, or even when the same model is deployed in different data center environments, so the accuracy of a model may vary across different disk populations.
Increasing reliability with maintenance tasks

In practice, erasure codes that can recover the data using any $k$ blocks of a reliability block are either computationally expensive (e.g. Reed-Solomon codes), or require the number of parity blocks, $m$, to be large compared to the $k$ data blocks that they are derived from, incurring significant storage overhead (e.g. consider the extreme case of plain mirroring where $k$ is equal to $m$) [88, 176]. Attempts have been made to solve this trade-off by proposing solutions that tolerate few errors (usually two per group), but are computationally efficient and incur small storage overheads [60, 175].

This shift in erasure code design, coupled with the fact that latent sector errors tend to appear in clusters, creates a fundamental problem in the scenario where data is not periodically verified. Then, errors will accumulate to the point where the available redundancy is inadequate for recovery. Another scenario involves double-parity inter-disk redundancy schemes, where the probabilities of two disks failing, or two sectors of the same inter-disk stripe exhibiting errors are small, but the probability of one disk failing, and another one exhibiting at least one latent sector error is considerably higher [94].

Scrubbing. To protect against data loss due to sector errors, modern commercial storage systems perform a periodical full-disk scan that aims to proactively detect and correct sector errors, a process called “disk scrubbing”. The frequency of these scans directly affects how reliable the system is, since errors will eventually cluster if left unattended [28], rendering redundancy schemes inadequate over time. Baker et al. [31] have shown, through the use of simple analytical models, that bi-weekly scrubs can achieve a Mean Time To Data Loss, that can be as high as 12.3 years for specific systems. The term “disk scrubbing” was first coined by Schwarz et al. [202], who present a scrubbing algorithm for archival systems, where scrubbing requests are piggybacked on normal traffic to avoid the cost of powering on disks just for scrubbing. The feasibility of the approach depends on the assumption that sufficient traffic will exist for full scrubs to complete over time. While scrubbers in production systems simply scan the disk sequentially, work by Oprea et al. [163] shows that an approach based on sampling can exploit LSE locality to reduce the error rate. The disk is separated into $R$ regions, each partitioned into $S$ segments. In each scrub interval, the scrubber begins by reading the first segment from each region ordered by the logical block numbers within each segment, then the second one, and so on. Other proposed algorithms focus on actions taken once an LSE is detected [167, 199], and Schroeder et al. [199] have used field data to verify the efficiency of the aforementioned scrubbing algorithms at finding errors.

Despite the fact that all the cited algorithms aim to reduce the mean time to error detection, their impact on the applications running on the system has not been verified in implementation. Additionally, the limited scope of disk sector scrubbing does not allow for the detection of inter-sector errors such as lost/torn and misdirected writes. To remedy that problem, scrubbing is recently becoming popular outside of the block layer, and is now implemented within filesystem code [38, 189]. This new type of scrubbing that attempts to correct errors at the granularity of blocks or files, has not yet been formally evaluated with regards to its impact on foreground applications that run concurrently. Our work, presented in Chapter 3, evaluates the impact of traditional sequential scrubbing, as well as the algorithm proposed by Oprea et al. [163], and provides a framework for the development and evaluation of other scrubbing algorithms. We have also evaluated the impact of scrubbing at the file system layer as part of the work presented in Chapter 4, and we find that it can increase latency by as much as $4 - 8x$ for user applications. This impact can be reduced to less than 2% when scrubbing requests are scheduled in the background, as shown in Figure 4.2.
Chapter 5. Related work

Write verification. Another approach for detecting silent corruption is write verification, or Read-after-Write (RaW). As writes are not verified on completion, a portion of the write errors may be detected and corrected successfully by verifying the data written on disk, with the data in the on-disk cache, or filesystem buffer cache. Traditionally, write verification is scheduled immediately after a write request completes, which is unattractive, because it degrades user performance by doubling the service time of those requests. Riska and Riedel [186] explored an approach that keeps written data in the on-disk cache, and delays verification requests to reduce their impact. Instead, verification is scheduled for a more opportune time when the disk is idle. Due to inevitable cache drops, their simulations show that their approach can verify more than 90%, but not all write requests, with up to 50MB of on-disk cache. In the context of distributed storage systems, Abd-El-Malek et al. [1] show how verification of distributed writes can be performed by any node in the system during idle time, and they point out that such intentional delays avoid verifying short-lived blocks. Our opportunistic approach can complement these methods, by allowing write verification to also take place when data is accessed by other maintenance tasks or user applications.

Data reconstruction. When a system that employs inter-disk redundancy experiences a disk failure, the failed disk’s contents need to be reconstructed in a spare disk using the available redundancy. In the case of RAID arrays, full disk scans need to be performed on all other disks in the array to compute the missing data. This can result in significant overhead for application requests [227]. An approach that leverages a type of opportunistic behavior, is that of Bachmat and Schindler [26], who propose scheduling reconstruction requests so that they piggyback on workload requests. However, the approach requires accurate information about hard disk geometry and monitoring of queues at the I/O scheduler level. Our opportunistic method requires minimal programming effort, and remains agnostic to device characteristics.

Another approach to data reconstruction is proposed by Sivathanu et al. [211] with D-GRAID, which uses filesystem-specific information to decide which blocks to reconstruct first, in order to maximize system availability. Since D-GRAID places semantically related blocks on the same device to increase availability, it needs to replicate them across devices to avoid creating bottlenecks that will severely impact RAID performance. Replication is only applied to popular files, which are detected via the collection of statistics. Our approaches, with regard to both I/O scheduling and opportunistic data processing are orthogonal to this work, and could be combined to further increase the efficiency of the reconstruction process.

5.1.2 Improving storage efficiency and security

Data deduplication. Data deduplication is a technique that aims to reduce storage needs by eliminating data redundancy. Only one copy of the data is retained on storage media, and redundant data is replaced with a pointer to the unique data copy. To achieve this, data is typically divided into small chunks and hash identifiers are used to represent and compare each chunk to those that have already been stored. The most popular approaches today include both file-level and block-level deduplication that is performed on both primary, and archival data. File-level deduplication has been found to be more effective in desktop environments [150], while the block-level approach has been shown to yield significant savings in enterprise environments [69]. In all cases, deduplication is performed using background jobs that scan the underlying volumes and identify duplicate files or blocks, which are then
replaced with block references. Orphaned references are detected by garbage collectors that also run in the background. Existing studies have focused on maximizing storage savings, however, and there is no work that attempts to quantify the impact of these background jobs on user applications. We believe that our opportunistic data processing model can be applied in the case of deduplication tasks, since a high degree of overlap is expected between their work and user applications. This is due to the expectation that these tasks will only have to operate on blocks that are modified by the workload.

**Layout organization.** Hard disk performance is inherently limited by the seek time and rotational delay required to find requested blocks. Therefore, organizing the layout of blocks so that related data end up close on disk can make a significant difference in performance and energy consumption. However, as filesystems age, files lose their sequentiality as they get modified by writes in random offsets, or due to bad allocation decisions which can become frequent as free space becomes sparse. Conventional filesystems have attempted to alleviate the performance problem through smarter allocation policies, but as Smith and Seltzer [214] have shown, allocation algorithms that work in some cases, may worsen the problem in others. Newer filesystems (e.g. ext4, Btrfs, ZFS) opt for online defragmentation instead [38, 189, 196]. Other approaches include: improving performance by utilizing free space close to frequently accessed blocks to store relevant block replicas [99], conserving energy by spreading files on disk arrays to service requests faster, and keep the devices powered on less [215], and using Log-structured Merge (LSM) trees to aggregate random updates in memory [209]. In all these cases, background jobs are required to be run, to either collect access pattern statistics, or mark modified blocks, in order to finally rearrange them at a later time. Such tasks, due to their natural dependence on the foreground workload, could benefit by the information provided through our Duet framework. We have already shown results on adapting the defragmentation task to our model. Another use of the events provided by Duet, however, could be to avoid duplicate work by monitoring file access patterns, e.g. to avoid rearranging files that experience mostly random accesses.

**Integrity checking.** Data integrity is verified by scanning data for patterns, or checking for unauthorized content modification. Processes that detect viruses belong in the former category, and attempt to detect short signatures that uniquely and efficiently identify malicious software. Virus scanners consist of two parts: a scanning engine and a component that feeds data to the scanning engine, and scans are either performed upon user request, or transparently when files are opened. Miretskiy et al. [157] propose a stackable filesystem that addresses the vulnerabilities of current techniques by scanning for viruses incrementally, upon access. The proposed filesystem, AVFS, is stateful, and remembers which pages of each file have been scanned for viruses. Once all pages have been scanned, the file is marked clean, and only future modification prompts further scanning. This approach, however, leaves the system vulnerable in the event that the device is tampered with, or when adversaries have raw access to its data. The solution for cases where devices can potentially be tampered with (e.g. cloud storage), are periodical scans that detect data modifications. Another situation where such scans become unavoidable, is when the virus signature database is updated, and all files must be checked again. To deal with this, Kim and Spafford [124] proposed Tripwire, a file integrity checker that scans a filesystem to detect changes in files and/or directories. Tripwire is an open-source widely used tool, that bases its success on a database of fixed-size file signatures generated using one-way hash functions. However, filesystem scans need to be triggered, incurring significant performance costs. While a blocking approach at the virtual file system layer, such as AVFS, can ensure that no accesses proceed without verification, the
Duet approach can act as a complementary mechanism that allows virus scanning work to be completed faster by piggybacking on the foreground workload or other maintenance tasks.

5.2 Identifying trends in enterprise backup systems

In addition to the maintenance presented in Section 5.1, storage system operators often need to rely on redundancy at various levels, to guarantee protection against storage errors, such as checksums and file integrity. Krioukov et al. [127] have shown that combining maintenance tasks with redundancy schemes, such as intra-disk and inter-disk redundancy, is necessary in order to adequately shield a storage system from data loss. A popular way to achieve data redundancy today is by taking data backups. Apart from our studies (see Chapter 2), however, little work exists in the literature to help us understand how backup systems operate, and their common failure modes. In Section 5.2.1, we provide an overview of the evolution of backup systems, and existing work that helps us understand how they are used in the field. Then, Section 5.2.2 reviews reliability literature that is relevant to our study of backup system failure modes.

5.2.1 Evolution of backup systems

Formally, backup is the process of making redundant copies of data, so that it can be retrieved if the original copy becomes unavailable. In the past 30 years, however, data growth coupled with capacity and bandwidth limitations have triggered a number of paradigm shifts in the way backup is performed. Recently, data growth trends have once again prompted efforts to rethink backup [2, 45, 64, 73, 95]. This section underlines the importance of field studies in this process.

In the early 1990s, backup consisted of using simple command-line tools to copy data to/from tape. A number of studies tested and outlined the shortcomings of these contemporary backup methods [125, 191, 261, 262]. The limitations of this approach, which included scaling, archive management, operating on online systems, and completion time, were subsequently addressed sufficiently by moving to a client-server backup model [35, 47, 54, 55]. In this model, job scheduling, policy configuration, and archive cataloging were all unified at the server side.

In the early 2000s, deduplicating storage systems were developed [178, 259], which removed data redundancy, lowering the cost of backup storage. Subsequently, Wallace et al. [243] published a study that aims to characterize backup storage characteristics by looking at the contents and workload of file systems that store images produced by backup applications such as NetBackup. A large body of work used their results to simulate deduplicating backup systems more realistically [79, 135, 136, 138, 139, 210, 225, 246], and was built on the motivation provided by the study’s results [81, 135, 137, 148, 213]. The authors analyze weekly reports from appliances, while we analyze reports from the backup application, which has visibility within the archives and the jobs that created them. However, the two studies overlap in three points. First, the deduplication ratios reported for backups confirm our findings. Second, we report backup data retention as a configuration parameter [9], while they report on file age, two distributions that overlap for popular values. Third, the average job sizes we report are 5-8 times smaller than the file sizes reported in their study, likely because they take into account all files in the file system storing the backup images. Overlaps between our study and previous work are also summarized in Table 2.7 of Section 2.3.1. Park and Lilja [168] have also published a study characterizing change
in full backup contents over time, using 30 weekly backup images from 6 machines. The deduplication ratios we report are in agreement with their findings as well.

Recently, an ongoing effort has been initiated in the industry to redefine enterprise data protection as a response to modern data growth rates and shorter backup windows [49, 61, 237]. Proposed deviations from the traditional model rely on data snapshots, trading management complexity for faster job completion rates [73], and a paradigm shift from backup to data protection policies, in which users specify constraints on data availability as opposed to backup frequency and scheduling [2]. The latter paradigm allows the system to make decisions on individual policy parameters that can increase global efficiency, while keeping misconfigurations to a minimum. In this direction, previous work leverages predictive analytics to configure backup systems [45, 64, 83]. We believe that all this work is promising, and that a study like ours, characterizing the configuration and evolution of backup systems over time could aid in developing new approaches and predictive models that ensure backup systems meet their goals timely, while efficiently utilizing their resources.

5.2.2 Understanding how enterprise backups fail

A number of studies previous to ours have examined the characteristics and prevalence of software and administration errors in a variety of contexts. In his seminal paper on system faults, Gray [89] reports that 25% of faults in high-end mainframes are due to software errors, while 42% are attributed to administrator errors. Similarly, Patterson et al. [170] observed that 8-34% of failures in telephone networks and Internet systems were due to software, while 51-59% were due to operator errors. Oppenheimer et al. [162] studied the failures of Internet Services and report that failures are due to software errors 33% of the time, while 57% are operator mistakes, and Nagaraja et al. [159] confirm these findings in a user study. Tang et al. [224] report that 16% of high-impact incidents at Facebook were due to misconfigurations, but do not provide a more detailed breakdown. Despite the existence of the aforementioned studies for other system types, our study is the first to consider backup systems, which have only been characterized so far with regards to their workloads [9, 168, 243]. It is worth noting that our breakdown of backup system errors by cause in Figure 2.13 of Section 2.2.3, matches closely the results reported in the literature for other system types.

Due to the prevalence of misconfigurations, several research efforts have put forth ideas to detect, diagnose, and automatically fix these errors. PeerPressure [245] uses a statistical approach to identify parameter errors for a single configuration, from a large set of configurations. In a similar manner, Yuan et al. [255] use support vector machines to detect anomalous patterns of system call sequences. CODE [256] extends this idea by identifying invariant configuration access rules. EnCore [258] adopts a learning approach guided by sample configurations, augmented with information on the execution context of the configurations. Chronus [250] retains disk state through periodical checkpoints, and traverses them to detect the configuration changes responsible for the misconfiguration. ConfAid [23] instruments application binaries to trace the configuration entry that is the source of the misconfiguration. AutoBash [22, 217] leverages carefully crafted bug-tracking predicates to detect deviations from healthy machines, and a speculative OS kernel to find a fix from a solution database using trial and error. Such approaches can benefit greatly from studies outlining the contributing factors to errors. Observations derived from such studies can be used as heuristics for learning approaches, and to prune the state space for data-flow analysis techniques. Furthermore, while these are successful reactive approaches to error occurrence, our full paper also examines the feasibility of proactive approaches, such as error prediction, which act
5.3 Scheduling maintenance work in the background

Background scheduling refers to techniques used to schedule requests originating in background tasks, such as the ones mentioned in the previous section. The goal of background scheduling is to mask the cost of servicing such requests so as not to impact the response times, or throughput, observed by the foreground application requests that are serviced as part of the system’s normal operation. The following sections describe the two methods of background scheduling that have been studied extensively in the literature: exploiting the characteristics of the hard disk drive servicing the requests, which is examined in Section 5.3.1, and leveraging the intervals when no foreground requests are outstanding and the device remains idle, which we look into in Section 5.3.2.

5.3.1 Micro-management at the I/O scheduler

A significant amount of work in the literature has focused on the design of algorithms that schedule hard disk I/O requests by exploiting the geometry and physical characteristics of the medium. In the instances that this method has been shown to apply, it can ensure significant bandwidth for background tasks. What follows are the highlights of relevant publications, and reasons that have made the approach harder to apply today.

Riedel et al. [184] introduce the idea of freeblock scheduling in a production OLTP system that is also running data mining tasks in the background, without impact on the foreground OLTP workload. The key insight of the approach is that while positioning the disk’s head for a foreground request, disk blocks passing under the disk head can be read for “free”, and passed to the background application for processing. Every time a foreground request is serviced, a sorting algorithm is invoked, consulting the list of pending background requests and the time between the current and next foreground request. The algorithm decides on a set of background requests that will not introduce seek time in excess of that required to service the foreground requests. The idea of freeblock scheduling was further developed by Lumb et al. [144], who discuss optimizations on the background request sorting algorithm that makes it feasible to run in real-time. They proceed to simulate the approach on a log-structured filesystem’s segment cleaner implementation [193], that kicks in when the number of empty segments drops below a certain threshold. Finally, Thereska et al. [226] define a programming framework for disk maintenance applications that is based on freeblock scheduling. Applications are expected to use this framework to make calls that explicitly state the disk activity that needs to be completed.

In order to accurately predict future positioning delays, freeblock scheduling requires detailed knowledge of disk performance attributes, such as layout algorithms and time-dependent mechanical positioning overheads, making the approach mainly applicable as a firmware implementation. Although attempts have been made to implement the approach outside the firmware [143], the error margins for timing mispredictions on modern disks are becoming significantly tighter due to multiple, more densely packed platters. At the same time, scheduling and prefetching algorithms within the disk firmware are becoming more complex and harder to manipulate [53]. Other attempts to avoid prediction errors include: preemptive disk scheduling [62], which breaks requests into smaller chunks (although seeking is harder to preempt and still dominates response time), explicitly extracting mapping information from disks that make it available [198], or even building models through close inspection of delays [251].
Unfortunately, the efficiency of freeblock scheduling is directly linked to the amount of background work available; more pending background requests increase the chances of finding pairs of foreground requests to fit them between. However, this results in higher computational and memory requirements for sorting the pending requests in time, making the approach harder to apply in modern systems where considerable amounts of storage are available. Instead, modern filesystems opt for online solutions that incrementally expose remaining work \[38, 189, 196\], operating on data on-demand (e.g. queuing files for defragmentation once they get fragmented). Although there is potential for exploiting synergies (assuming that overlapping block requests happen to be emitted by the applications at the same time), this is more of a side-effect of the freeblock scheduling approach, rather than a clear goal of the stated work. Finally, due to the opportunistic nature of the freeblock scheduling algorithm, disk blocks are likely to be fetched in random order, so background applications are expected to be able to process disk blocks as they arrive. This makes the approach hard to apply for tasks that operate in larger groups of data, such as files or filesystem blocks.

All the approaches in this section, however, make use of prefetching and merging, mechanisms that should ideally be avoided for maintenance work, to avoid polluting the device and page caches with data that were not requested by user applications. Furthermore, in some cases such as scrubbing, it is necessary to guarantee that sector contents were verified from the medium’s surface (rather than a cache) at request execution time.

### 5.3.2 Exploiting idleness

In the field, it is common practice to manually schedule maintenance tasks during times when the system is expected to be otherwise idle. To automate this process, Golding et al. \[84\] investigate several idle-time detection algorithms, and define a number of metrics that are used to derive an extensive taxonomy. The algorithms are simulated against workload traces, however the evaluation is based on fixed parameters that worked specifically for the given workloads.

Bachmat and Schindler \[26\] analyze two algorithms that are more adaptive when scheduling background work: one that greedily services background requests sequentially from the beginning to the end of the disk, and one that services requests at random, by piggybacking on the foreground workload. The background requests are trickled into the scheduler queue one by one, or when the queue is empty of foreground requests. They show that by servicing requests greedily seek penalties are minimized.

Another approach is limiting the bandwidth allocated to background tasks. There exists a rich body of work on weighted allocation of I/O resources to individual applications. Techniques such as Stonehenge \[100\] and SFQ(D) \[113\] are based on fair-queueing algorithms originally proposed for network bandwidth allocation, such as WFQ \[58\] and SFQ \[87\], but they employ optimizations that allow them to achieve higher efficiency in the context of storage systems. Wang et al. have also proposed DSFQ \[248\], a scheduler focusing on proportional I/O resource allocation in distributed storage systems, that is based on cooperation between the underlying storage device and storage clients. Wachs et al. \[242\] investigate an approach that limits applications’ access to filesystem cache and disk time. Their framework, Argon, relies on a per-task threshold that defines the acceptable slowdown in throughput. This slowdown is relative to the computed throughput of the task, were it assigned the same fraction of the device exclusively. Their user-space scheduler partitions the available cache and limits the amount of disk time used to achieve the desired throughput goal, but gives no guarantees on response times. Similarly, Povzner et al. \[177\] present Fahrrad, a framework that allows for weighted allocation of I/O resources.
by time-slicing disk accesses to reduce interference across streams accessing the same device. Seelam and Teller [204] use a similar time-slicing technique in VIO, a scheduler providing performance isolation in virtualized storage systems. Finally, another approach relies on manipulating queue depth to meet latency bounds [142], at the cost of efficiency with regards to device utilization [257].

More recently, Riska and Riedel [185, 187] analyzed several disk workloads, and found that the distributions of idle interval lengths are consistently characterized by heavy, long tails. This means that a small fraction of the idle intervals are really long, and make up the majority of the total idle time in the system. This is a desirable characteristic, since long idle periods are simple to predict, and can even be used to improve the system’s energy efficiency. Jejurikar and Gupta [111] demonstrate that procrastination scheduling, an approach that delays pending work while respecting task deadlines, can result in energy gains of up to 10%. In a work more related to ours, Mi et al. [151, 152] present a scheduling framework for low priority tasks. The framework monitors the system and builds a histogram of idle interval lengths, that is then used to determine the duration of future idle intervals. Based on that histogram, background requests are scheduled only when it is expected that the foreground workload will not be slowed down beyond a certain degree. The work of Mi et al. [151, 152], similar to the literature reviewed in the previous subsection, gives no guarantees on the time or order that background requests will be scheduled. On the other hand, the cited works focus on the specifics of scheduling background requests, without taking into account the characteristics of the background tasks serviced, and potential overlap with other tasks or applications running in the system. The approaches presented in this thesis leverage such information to reduce the amount of remaining work. We further schedule remaining work so that its impact on user applications is minimized.

5.4 Opportunistic data processing

Our study of 40,000 enterprise backup systems, monitored over a span of 3 years, found that full backups are performed frequently: 28% of systems conduct one every 1-3 days, 44% perform them every 3-6 days, and only 17% of systems perform them weekly [9]. Moreover, even in the case of systems that perform full backups weekly or less frequently, these backups are complemented with daily incremental backups. In our study, we also report that administrators simply use the default scheduling windows when configuring backup policies, causing backup jobs to execute in bursts. Anti-virus scans in virtual machines have been shown to cause I/O storms for similar reasons [212]. These observations suggest that maintenance work is performed frequently and concurrently, where Duet’s opportunistic approach is expected to have the most benefits.

As we have discussed in previous sections, existing approaches for reducing the impact of maintenance tasks primarily focus on scheduling I/O requests [144, 226]. They require specifying the requests (in terms of sets of disk locations that will be read or written) sufficiently in advance to the scheduler, which complicates programming and limits flexibility. These approaches help amortize the cost of seeks and rotational delays but do not necessarily reduce the number of I/O operations. Consider two hypothetical tasks, one that traverses the file system in depth-first order, and the other in breadth-first order. If these tasks are run concurrently, even careful scheduling of I/O requests may not provide much benefit. Even if the two tasks traverse the file system in the same order, but are staggered in time, then the benefits of scheduling will be limited. These examples argue for out-of-order processing at the level of the maintenance application itself, which helps with both of these issues. Our opportunistic approach
provides hints to applications to enable efficient out-of-order processing.

With scheduling approaches, task-specific deadlock prevention mechanisms are needed for handling inter-block dependencies [226]. For example, if a block needs to be moved to a location containing live data, then the live data is copied to a persistent staging area until it can be moved to its own new location. However, deadlocks can occur if the staging area fills up with blocks that have unresolved dependencies. Our approach sidesteps this issue because it uses hints, and out-of-order processing is performed within the maintenance application rather than at the scheduler level. In particular, Duet has no knowledge of dependencies and hence does not require staging.

Our work, which aims to reduce I/O accesses, is orthogonal to general I/O scheduling schemes, such as anticipatory scheduling [109], idle-time scheduling [66] and others [26, 84, 151]. We can use any of these to run maintenance tasks, but we found experimentally that running maintenance tasks with an idle-time scheduler [84] minimizes impact on workloads. The scheduling algorithm does not reduce the amount of maintenance work, however, so the challenges with meeting maintenance goals still persist [9, 157, 237].

Sharing work across some maintenance tasks is supported by the Simpana suite from CommVault [49], which performs a single file system pass and delivers data to “converged” backup, archive and reporting tasks that are part of the same application. Our opportunistic work model supports existing maintenance tasks that have been developed independently, both at the user and kernel level. We are also able to take advantage of data cached as a result of workload accesses.

In the context of distributed systems, Ananthanarayanan et al. have presented PACMan [13], a framework that helps reduce job completion times in a cluster environment with a global cache replacement policy that coordinates data that is cached in memory across nodes. On a similar note, Venkataraman et al. designed KMN [239] for a new class of jobs, such as machine learning algorithms, that can compute on any subset of the input dataset. KMN uses this property to co-locate tasks with input data, thus reducing the time taken by tasks to read their inputs. Anastasiadis et al. [14] explored an application-level block reordering technique that can reduce server disk traffic, which specializes in large content files that are shared by concurrent clients. Our work focuses on long running (large) tasks, whose data may not fit in caches. We present a framework that provides any task with the necessary information to make the best use of the current cached data, by modifying the order in which tasks process data. However, we expect that informed cache replacement, such as in the case of PACMan, will provide us additional benefits.

The database community has investigated data-driven approaches, with systems actively scanning in-memory data and invoking interested queries [6, 235], or pushing data onto processors and allowing any interested computation to process it [21]. These approaches take advantage of declarative database queries to perform out-of-order processing. We use polling and hints so that tasks can perform out-of-order processing, instead of using a pure data-driven approach. The latter may be harder to retrofit in existing, imperative tasks that impose specific ordering requirements.
Chapter 6

Ongoing and future work

We are currently working on extending the work presented in this thesis. This chapter outlines both ongoing, and future work planned in three immediate directions. First, we are working on the application of our opportunistic work model in the context of distributed computational frameworks, an extension that shows promising preliminary results as we have shown in Section 4.5.1. We are also continuing our work augmenting applications that rely on file notification frameworks, so that they leverage information provided by Duet. Our current results on the Seafile file synchronization application, as shown in Section 4.5.2, are encouraging. Finally, we are working on leveraging the observations in our studies of enterprise backup systems to improve the efficiency and robustness of next-generation backup software. We describe our intended next steps in each of these directions in detail, in the remainder of this chapter.

6.1 Opportunistic computational frameworks

In Section 4.5.1 we presented Quartet, a series of modifications mostly concentrated in the scheduling components of Hadoop and Spark. While our preliminary results seem promising, we have still to utilize the full potential of Duet information and evaluate its effect on more realistic scenarios.

Currently, we are using Duet events to determine what fractions of individual HDFS blocks are present in the operating system caches of our cluster’s nodes. We can further utilize this information to determine the rate of churn for data at individual node caches. By leveraging Duet events in this way, we can predict which multi-page HDFS blocks are in the process of being added, or evicted from the cache. This will allow the scheduler to make the decision to prioritize tasks interested in said blocks earlier. In some frameworks, such as Hadoop, there is a significant delay to deploying task containers after a scheduling decision has been made. Therefore, making scheduling decisions in a more timely fashion can reduce the probability of a given block having been partially, or fully evicted by the time interested tasks process its data. Another use of information on cache churn rates is to help tune the framework’s scheduling algorithms with regards to the operating system’s caching policy, which may differ across nodes. One such example is inferring the average time that a block remains cache-resident on individual nodes. Finally, Duet events can also be used for trace collection on cache performance for future analysis.

While our evaluation of Quartet is promising, we have only experimented with toy scenarios where identical jobs run back-to-back. Previous work by Ren et al. [183] has shown that such high levels of
data reuse for a given user can be encountered in the field, however they are expected to be part of a larger, multi-user workload. Unfortunately, traces that are currently available for MapReduce workloads, or distributed computational frameworks in general, lack the necessary information to infer data reuse levels. Specifically, existing traces either report job scheduling information without inclusion of data accesses, or sample HDFS block accesses without linking them to job scheduling information or specific replicas (and by extension, devices). We are approaching this limitation by targeting the augmentation of specific job types, such as machine learning jobs. In the case of certain machine learning techniques, such as model ensembles, individual models need to be trained on overlapping subsets of the input. Often, analysts are also required to explore a large state space of parameters, to find the optimal values for their dataset. Finally, we envision Quartet being used in large-scale data centers where it could be most effective. To enable this, however, we need to ensure that Quartet components can scale efficiently without prohibitive overheads. We are currently exploring the use of Quartet in these scenarios.

6.2 Enhancing file notification frameworks

In Section 4.5.2 we elaborated on the benefits of using Duet in conjunction with file notification frameworks. Our exploration of this area is somewhat limited, as we currently only consider a file synchronization application. File notification frameworks, however, are used by a plethora of applications that react to their events differently: whether an event occurs on a configuration file, a critical system file, or a log, and depending on the application’s type, the response time to the event can significantly affect performance. We have conclusively demonstrated this in the case of Seafile’s file chunking. We plan to examine a variety of application types using file notification frameworks, in order to gain a better understanding of how Duet could help them improve their efficiency.

In the case of our Seafile implementation, we are currently faced with a number of open questions. First, Duet does not require applications to recursively add watches on all watched directories in the repository. Seafile today is required to traverse the entire repository as part of its bootstrapping process, incurring significant I/O. We have not measured the benefit of Duet in this scenario. Second, Seafile’s synchronization performance depends on file contents, as only unique chunks are sent over the network. Thus, we have chosen to populate our files with random data, in order to introduce high entropy and evaluate the effect of the chunking algorithm in scenarios with significant network traffic. We have not studied, however, the effect of this high entropy in file contents, on other components of the system. We are planning to evaluate Seafile using macrobenchmark workloads on the repository data, as well as populate files with content derived from real-world entropy distributions.

6.3 Next-generation backup systems

In Chapter 2 we analyzed an extensive dataset of telemetry reports from enterprise backup systems. In Section 2.2 we showed that job errors are prevalent and mostly attributed to misconfigurations. Fortunately, we showed that errors occurring frequently are not diverse in nature, with 10 error codes accounting for over 78% of job errors. In Section 2.3 we also studied the way these systems are configured and evolve over time. We find that with regards to job scheduling, the popularity of default values can have an adverse effect on the efficiency of the system by creating bursty workloads. We also showed that backup systems grow in bursts, and frequent, deduplicated full backups are preferred as a
response to data growth. This section looks at potential avenues for future research that would enable next generations of backup systems to achieve higher efficiency and lower job failure rates.

**Configuration automation.** Our results confirm a well-known trend in the software reliability literature: the prevalence of misconfiguration errors. In the case of a maintenance task, backup, we show that misconfigurations can incur failures that could potentially jeopardize the guarantees provided. A small body of prior work attempt to remedy this by enabling self-configured systems, where software parameters are automatically adjusted to resolve misconfigurations. AutoBash [217] leverages a speculative OS kernel to fix misconfigurations. Chronus [250] makes use of checkpointing and rollback to detect the last working configuration. KarDo [128] takes a machine learning approach to learn the steps required to resolve the issue from a crowd-sourced collection of solutions, supporting heterogeneous environments as well. NetPrints [3] collects examples of good and bad network configurations, and generates a decision tree to determine the set of configuration changes required to transition the system to a good state. One of the challenges that these techniques face, is their timeliness in providing solutions. In some contexts, such as that of maintenance tasks, allowing misconfigurations to remain latent can adversely affect the system’s availability, reliability, security, etc. In our full paper [10] we further analyze our dataset and provide observations that could be used as heuristics to speed up the search for the root causes of misconfigurations in enterprise backup systems. Other approaches attempt to alleviate performance and availability problems of backup systems by using historical data to perform automated storage capacity planning [45], data prefetching, and network scheduling [83]. Our findings, both here and in Section 2.2 support this line for work. Ensuring maintenance work is efficiently scheduled can reduce completion times and job failure rates, thus avoiding work repetition. Furthermore, we have shown that backup domains grow in bursts, and misconfigurations are common. Enabling configuration to be automated will, therefore, prevent clients from being left unprotected, jobs being scheduled in bursts, and users not being warned about imminent problems. In Chapter 3 we demonstrated the effect of maintenance configuration on efficiency, and the importance of determining configuration parameters in an automated manner.

**Configuration validation.** The major research efforts towards addressing misconfiguration errors focus on detecting, diagnosing, and troubleshooting these issues. While this provides a remedy by finding the root causes and solutions for individual cases, it does not spare users from the frustration of having to deal with these errors in the first place. Fortunately, another line of research focuses on improving the design of configuration interfaces, improving the resilience of the systems towards misconfigurations. Some of the existing work includes the extraction of configuration parameters and their types [180], statically analyzing the code to infer constraints on configuration variables [252], permuting valid configuration settings [121], testing operator actions in sandboxes before making them visible [159], storing application state to seamlessly undo and replay events in the case of errors [43], and using a declarative language to express configuration specifications [101]. We find that the configuration complexity of backup policies is strongly correlated to failures [10]. Thus, we expect that working on automating even a part of the configuration process will both increase the efficiency of configuration validation and reduce failure rates.

**Work reduction.** We have shown that a common job failure mode is that jobs fail to get scheduled because the scheduled maintenance time is insufficient. We expect this problem to worsen in systems that rely on multiple maintenance tasks, apart from backups. To alleviate this, maintenance work
will have to be reduced and performed more efficiently. In Chapter 4, we presented a framework that allows concurrently running maintenance tasks to collaborate in order to reduce the total I/O required, effectively reducing their runtime.

**Content-aware backups.** Backup strategies can generate data at a rate up to 5 times higher than production data growth [2]. This is due to the practice of creating multiple copies and backing up temporary files used for test-and-development or data analytics processes, such as the Shuffle stage of MapReduce tasks [46]. Depending on the storage interface used, it might be more efficient to recompute these datasets rather than restoring them from backup storage. This problem generalizes to other tasks, because the generated backup data will need to be maintained by other processes over time, such as scrubbing the data for errors, or enabling backup software to detect data changes since the last backup among PBs of data and billions of files [102]. Furthermore, we have shown that backup data growth is significant, even for deduplicated primary copies. By augmenting maintenance tasks to account for data types and modification events, we can potentially reduce the time needed to complete maintenance work.

**Accident insurance.** Most recovery operations in our dataset appear to be small in both the number of files and bytes they recover, compared to their respective backups. This result suggests that recovery operations are mostly triggered to restore a few files, or to test the integrity of backup images. This motivates us to re-examine the requirement of instant recovery for backup systems as a problem of determining which data is more likely to be recovered, and storing it closer to clients [135, 140].

**Deduplication.** Our findings confirm the efficiency of deduplication at reducing backup image sizes. We further show that in many systems, incremental backups are replaced by frequent full, deduplicated backups. This is likely due to the adoption of deduplication, which improves on incremental backups by looking for duplicates across all backup data in the domain. To completely replace incremental backups, however, it is necessary to improve on the time required to restore the original data from deduplicated storage, which directly affects recovery times. Currently, this is an area of active research [80, 115, 138, 161].
Chapter 7

Conclusion

The way we interact with information has recently undergone dramatic change. This change was led by the availability of centralized storage in public and private clouds, which enabled large amounts of data to be stored inexpensively for extended periods of time. As a result of this change, enterprises have been able to extract more insight from their data, which is ushering in an era of reliance on data analytics. Meanwhile, exponential data growth rates [208] make it increasingly challenging for cloud storage providers to ensure that data is stored reliably, efficiently, and securely. At this scale, potential data loss events and leaks can impact information of varying importance, from personal communications to crucial corporate data, affecting the lives of large groups of people, and business operation. At the same time, cloud providers are often required to store data efficiently in order to balance guarantees on tail access latencies [42] and storage costs.

To respond to these requirements, modern storage systems employ maintenance tasks that provide a host of guarantees, from security and reliability to performance, which have been extensively covered in this thesis. To be successful, however, these tasks must perform their maintenance operations ahead of an undetermined deadline: either before applications access the data, or before a disaster occurs. We have shown that predicting this deadline can be challenging, especially without proper hardware support [145, 174]. Our study of more than 270,000 hard disk drives deployed across dozens of Google’s datacenters shows that both whole and partial disk failure rates can differ across drive models and data center operating environments. Furthermore, we provide evidence that periodic maintenance does not incur additional failures, contrary to what is commonly assumed [202].

One important challenge with maintenance work, is that it can significantly impact the performance of user applications, by increasing the latency of operations that are performed at the same time as maintenance work. To lessen the impact of maintenance today, storage operators perform maintenance work during scheduled downtime periods, when normal system operation is suspended. In a study we conducted across 20,000 enterprise backup systems deployed by Veritas customers, we find that 1 in 10 jobs fail to complete. We monitored these systems over a span of 3 years to collect data for 775 million jobs, and found that one of the common job failure modes is lack of sufficient system downtime to complete the backup jobs.

An alternative approach that allows maintenance work to be complete without affecting user applications involves scheduling maintenance requests while the device is otherwise idle [84, 151, 152]. In order to achieve this, device idleness must be accurately predicted. In order to understand the characteristics
of idleness, we performed a detailed statistical analysis of 77 publicly available I/O traces of a diverse collection of disk workloads. Using insights from our analysis we defined several predictors for idleness, and evaluated them using our trace data. We find that the simplest predictor performs optimally, where a threshold is used to distinguish long idle intervals that should be used for maintenance work. By utilizing long enough intervals, the predictor allows the majority of total idle time to be leveraged without incurring more slowdown than a predefined goal. We tested our efficiency by implementing scrubbing algorithm both at the user and kernel levels. Using our scrubber implementations we performed the first experimental comparison between the sequential and staggered scrubbing algorithms. While sequential scrubbing is the approach that is currently used in production systems, we find that when staggered scrubbing parameters are tuned properly it can achieve the same or better scrub throughput as a sequential scrubber, without additional penalty to user applications. This is important, because staggered scrubbing has the ability to find LSEs faster by probing different regions across the drive’s surface [163].

When idle time is insufficient, it can be challenging to schedule maintenance work without impacting user applications. This problem occurs as idle time is becoming increasingly scarce in shared environments, such as the cloud, where high consolidation ratios are the goal by definition. Moreover, maintenance tasks typically access large amounts of data that does not fit easily in memory, and they are usually executed on a frequent schedule. We investigated an extensive dataset of 40,000 enterprise backup systems of Symantec customers over the span of 3 years, and we found that the majority of full backups occur as often as every 1-4 days. We also report large job sizes, despite the fact that backup job data is almost always deduplicated. This abundance of maintenance work is further exacerbated for systems that depend on multiple maintenance tasks.

Today, maintenance tasks are implemented and optimized in isolation. At the same time, these tasks are characterized by overlap in the work they perform, due to their wide application on available data. To leverage such synergistic behaviour, we have presented a model that allows storage maintenance to be performed opportunistically based on data cached in memory. Maintenance tasks perform out-of-order operations on this data, reducing the total I/O needed. We designed and built Duet, a framework that provides notifications about page-level events to tasks. This granularity works well for both the block or file granularity processing performed by maintenance tasks, requiring relatively small task changes to perform out-of-order processing. Through our evaluation we showed that opportunistic maintenance tasks require less I/O and complete faster, with benefits that can increase as more tasks are executed concurrently.

The findings presented in this thesis suggest that maintenance work does not have to be relegated to a maintenance window, which is hard to schedule because idle times are unpredictable and may not be sufficient for the work needed. Instead, maintenance work should be done at low priority, continuously and synergistically with other workloads to minimize its impact. We believe that the proposed approaches can be generalized to other applications with a similar declarative nature to maintenance tasks, such as data analytics processes, and we are currently working on extending our methods to different applications. Specifically, we have shown promising preliminary results when our opportunistic model is applied to distributed computational frameworks, or as an extension of the traditional file notification frameworks. In conclusion, as people and corporations are becoming increasingly dependent on the insights provided by data analytics tasks, and the guarantees provided by maintenance tasks, we expect our work to become more relevant in the future.
Bibliography


[73] EMC Corporation. EMC ProtectPoint: Protection Software Enabling Direct Backup from Primary Storage to Protection Storage, 2014.


[86] Google Official Blog. We knew the web was big... https://googleblog.blogspot.ca/2008/07/we-knew-web-was-big.html, July 2008.


