Towards a Flexible High-efficiency Storage System for Containerized Applications

Nannan Zhao

Dissertation submitted to the Faculty of the Virginia Polytechnic Institute and State University in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Computer Science and Applications

Ali R. Butt, Chair
Vasily Tarasov
Na Meng
Dongyoon Lee
Changhee Jung

August 24, 2020
Blacksburg, Virginia

Keywords: Containers, Distributed storage systems, Deduplication, Wear Balancing, Flash memory, Docker registry, Docker images
Copyright 2021, Nannan Zhao
Towards a Flexible High-efficiency Storage System for Containerized Applications  
Nannan Zhao  
(ABSTRACT)

Due to their tight isolation, low overhead, and efficient packaging of the execution environment, Docker containers have become a prominent solution for deploying modern applications. Consequently, a large amount of Docker images are created and this massive image dataset presents challenges to the registry and container storage infrastructure and so far has remained a largely unexplored area. Hence, there is a need of docker image characterization that can help optimize and improve the storage systems for containerized applications. Moreover, existing deduplication techniques significantly degrade the performance of registries, which will slow down the container startup time. Therefore, there is growing demand for high storage efficiency and high-performance registry storage systems. Last but not least, different storage systems can be integrated with containers as backend storage systems and provide persistent storage for containerized applications. So, it is important to analyze the performance of different backend storage systems and storage drivers and draw out the implications for container storage system design. These above observations and challenges motivate my dissertation.

In this dissertation, we aim to improve the flexibility, performance, and efficiency of the storage systems for containerized applications. To this end, we focus on the following three important aspects: Docker images, Docker registry storage system, and Docker container storage drivers with their backend storage systems. Specifically, this dissertation adopts three steps: (1) analyzing the Docker image dataset; (2) deriving the design implications; (3) designing a new storage framework for Docker registries and propose different optimizations for container storage systems.

In the first part of this dissertation (Chapter 3), we analyze over 167TB of uncompressed Docker Hub images, characterize them using multiple metrics and evaluate the potential of file level deduplication in Docker Hub. In the second part of this dissertation (Chapter 4), we conduct a comprehensive performance analysis of container storage systems based on the key insights from our image characterizations, and derive several design implications. In the third part of this dissertation (Chapter 5), we propose DupHunter, a new Docker registry architecture, which not only natively deduplicates layers for space savings but also reduces layer restore overhead. DupHunter supports several configurable deduplication modes, which provide different levels of storage efficiency, durability, and performance, to support a range of uses. In the fourth part of this dissertation (Chapter 6), we explore an innovative holistic approach, Chameleon, that employs data redundancy techniques such as replication and erasure-coding, coupled with endurance-aware write offloading, to mitigate wear level imbalance in distributed SSD-based storage systems. This high-performance flash cluster can be used for registries to speedup performance.
Towards a Flexible High-efficiency Storage System for Containerized Applications

Nannan Zhao

(GENERAL AUDIENCE ABSTRACT)

The amount of Docker images stored in Docker registries is increasing rapidly and present challenges for the underlying storage infrastructures. Before we do any optimizations for the storage system, we should first analyze this big Docker image dataset. To this end, in this dissertation we perform the first large-scale characterization and redundancy analysis of the images and layers stored in the Docker Hub registry. Based on the findings, this dissertation presents a series of practical and efficient techniques, algorithms, optimizations to achieve high performance and flexibility, and space-efficient storage system for containerized applications. The experimental evaluation demonstrates the effectiveness of our optimizations and techniques to make storage systems flexible and space-efficacy.
Dedicated to my parents Yanfang Zhang and Deyu Zhao.
Acknowledgments

This dissertation would not have been possible without the support of many people who have become an integral part of my life. I would like to take this time to express my gratitude to these awesome individuals.

First, I would like to thank my advisor Ali R. Butt for his constant support, guidance, and encouragement. Dr. Ali gave me sufficient space during my Ph.D to pursue my research ideas and become an independent researcher. Dr. Ali has taught me everything that was needed to be successful in my doctoral program such as, how to write a good research paper, how to deliver a good presentation, how to balance research work and courses, how to talk with peers and build a productive collaboration relationship, etc. Without Dr. Ali, I would have never achieved this goal.

I would also like to thank my Ph.D committee members: Dr. Vasily Tarasov, Professors Na Meng, Dongyoon Lee, and Changhee Jung, for their insightful comments and valuable feedback on my dissertation during my Ph.D.

I have been fortunate to work with many talented researchers and faculty members at Virginia Tech and other institutes. I am grateful to Xubin He (Temple University), Feiyi Wang (Oak Ridge National Laboratory) for their valuable insights for Chameleon project. I am grateful to Vasily Tarasov, Ali Anwar, Lukas Rupprecht, and Dimitrios Skourtis (IBM Research Almaden) for their guidance on Docker image analysis, Slimmer, and Duphunter projects. I am grateful to Professor Xun Jian for his guidance on Memory compression project. I am grateful to Amit S. Warke and Mohamed Mohamed (IBM Research Almaden) for their help on Docker image analysis and Slimmer project. I am grateful to Dean Hildebrand (Google) for his help in Slimer project. I am really grateful to Mohammed Salman and Yue Cheng for their help in Chameleon project. Last but not least, I am really thankful to Hadeel Albahar and Subil Abraham for their help in Duphunter project.

My work as a Ph.D. student was made easier, and enjoyable, due to working with a wonderful group of labmates at DSSL. During my first year of my Ph.D at VT, Ali Anwar, Yue Cheng, Mohammed Salman, and Safdar Iqbal helped me a lot on research projects, TA works, and other stuff. Without them, my early years of grad school life would have been very boring.
and difficult. After they graduated, I am extremely fortunate to have new labmates Hadeel Albahar, Subil Abraham, and Keren Chen to work on new research projects, brainstorm new ideas, and discuss stuff going around. I am extremely thankful to have Bharti Wadhwa, Arnab K. Paul, and Jingoo Han as my labmates throughout my Ph.D. life at VT. I will always be grateful for their support and encouragement. I am extremely fortunate to have worked with a wonderful group of labmates at DSSL and Professor Jian’s group: Luna, Sangeetha, Michael, Jamal, Muhammad Laghari, and Gagandeep Panwar. The collaborations and social interactions have enriched my grad school life.

I made several friends during my stay at Blacksburg and my internship at IBM research. I met two wonderful roommates when I first came to Blacksburg, Wenjun Hu and Qingyang Chen. They helped me to settle down and taught me to cook food. I will always cherish the good memories of cooking sessions. I met Jianling Wang in the hotel on the first day of my internship at IBM research. She was a very nice roommate to me. I will always cherish the good memories of California, shopping, food, and wonderful road trips.

In closing, I am immensely grateful to the most important people - my family, without whom this Ph.D. would have been lightweight and meaningless. My dad, Deyu Zhao, always supported and encouraged me. My mom, Yanfang Zhang, always took care of me and encouraged me to follow my dream and ambitions. They are my heroes and their love has helped me go through many difficulties. My boyfriend, Muhui Lin, is my constant source of enthusiasm, creativity, and love. I thank him for always supporting me and making me a better person. I would like to thank my grandma for her love throughout my life. I would like to thank my brother and my sister-in-law for providing constant support and help. I dedicate this dissertation to my family and friends.

Funding Acknowledgment My research was supported by National Science Foundation (CNS-1405697, CNS-1615411, CNS1565314/1838271, CCF-1919113, OAC-2004751, CNS-1565314, CCF-1547804, CNS-1702474, and CCF-1717660), CS at Virginia Tech, CIS at Temple, and IBM Research-Almaden. Many thanks to all for making this dissertation possible.

Declaration of Collaboration In addition to my advisor Ali R. Butt, this dissertation has benefited from many collaborators, especially:

- Vasily Tarasov, Ali Anwar, Lukas Rupprecht, Dimitrios Skourtis, Amit S. Warke, Mohamed Mohamed, and Dean Hildebrand contributed to the Docker image analysis in Chapter 3 of this dissertation, and have been part of multiple joint papers [193, 194].

- Vasily Tarasov, Hadeel Albahar, Ali Anwar, Lukas Rupprecht, Dimitrios Skourtis, Arnab K. Paul, Keren Chen, and Mohamed Mohamed contributed to the work included in Chapter 4.
• Hadeel Albahar, Subil Abraham, Keren Chen, Vasily Tarasov, Dimitrios Skourtis, Lukas Rupprecht, and Ali Anwar contributed to [198] in Chapter 5.

• Ali Anwar, Yue Cheng, Mohammed Salman, Xubin He, and Feiyi Wang contributed to [195] in Chapter 6.
Contents

List of Figures ........................................... xii

List of Tables ............................................ xv

1 Introduction ............................................. 1

1.1 Motivation ............................................. 1

1.1.1 Large-Scale Analysis of the Docker Hub Dataset ................. 2

1.1.2 Design Implications for Container Storage Systems via Performance Analysis ............................................. 3

1.1.3 Flexible High-performance Deduplication for Docker Registries .... 3

1.1.4 Adaptive Wear Balancer for Flash Clusters .......................... 4

1.2 Research Contributions .................................... 4

1.3 Dissertation Organization .................................. 6

2 Background .............................................. 7

2.1 Docker Registry Storage Systems ............................. 7

2.2 Deduplication in Storage Systems .............................. 8

2.3 SSD-based storage cluster .................................. 8

3 Large-Scale Analysis of Docker Images ......................... 10

3.1 Introduction ............................................. 10

3.2 Background and Methodology ................................ 11

3.2.1 Background ........................................... 11
3.2.2 Methodology .................................................. 13

3.3 Dataset Characterization ........................................... 15
  3.3.1 Layers and images ........................................... 15
  3.3.2 Files ......................................................... 18

3.4 Deduplication Analysis ............................................ 21
  3.4.1 File level deduplication ..................................... 21
  3.4.2 Deduplication by file types ................................ 23

3.5 Chapter summary ................................................... 25

4 Design Implications for Container Storage Systems via Performance Analysis .................. 26
  4.1 Introduction ................................................... 26
  4.2 Background .................................................... 27
    4.2.1 Container storage drivers ............................... 27
    4.2.2 Related work on storage drivers ....................... 27
  4.3 Layer Pulling Latency Analysis ................................. 28
    4.3.1 Performance breakdown ................................... 29
    4.3.2 Storage option impact on packing and unpacking .......... 31
    4.3.3 Concurrency impact ....................................... 32
  4.4 Container Storage Driver Performance Analysis .................. 32
    4.4.1 Small I/O requests ....................................... 33
    4.4.2 CoW performance .......................................... 35
    4.4.3 Concurrency impact ....................................... 36
    4.4.4 Analyses Summary ......................................... 37
  4.5 Registry with Deduplication Support ............................... 37
    4.5.1 Design ................................................... 37
    4.5.2 Performance evaluation ................................... 38
    4.5.3 Enhancements ............................................. 40
  4.6 Chapter Summary ................................................ 40
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td><strong>Flexible High-Performance Deduplication for Docker Registries</strong></td>
<td>42</td>
</tr>
<tr>
<td>5.1</td>
<td>Introduction</td>
<td>42</td>
</tr>
<tr>
<td>5.2</td>
<td>Background</td>
<td>44</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Docker Registry</td>
<td>44</td>
</tr>
<tr>
<td>5.2.2</td>
<td>Deduplication</td>
<td>45</td>
</tr>
<tr>
<td>5.3</td>
<td>Motivating Observations</td>
<td>46</td>
</tr>
<tr>
<td>5.3.1</td>
<td>Redundancy in Container Images</td>
<td>46</td>
</tr>
<tr>
<td>5.3.2</td>
<td>Drawbacks of Existing Technologies</td>
<td>47</td>
</tr>
<tr>
<td>5.3.3</td>
<td>Predictable User Access Patterns</td>
<td>48</td>
</tr>
<tr>
<td>5.4</td>
<td>Duphunter Design</td>
<td>49</td>
</tr>
<tr>
<td>5.4.1</td>
<td>Overview</td>
<td>50</td>
</tr>
<tr>
<td>5.4.2</td>
<td>Deduplicating Layers</td>
<td>52</td>
</tr>
<tr>
<td>5.4.3</td>
<td>Restoring Layers</td>
<td>53</td>
</tr>
<tr>
<td>5.4.4</td>
<td>Caching and Preconstructing Layers</td>
<td>54</td>
</tr>
<tr>
<td>5.4.5</td>
<td>Discussion</td>
<td>56</td>
</tr>
<tr>
<td>5.5</td>
<td>Evaluation</td>
<td>56</td>
</tr>
<tr>
<td>5.5.1</td>
<td>Evaluation Setup</td>
<td>56</td>
</tr>
<tr>
<td>5.5.2</td>
<td>Deduplication Ratio vs. Performance</td>
<td>58</td>
</tr>
<tr>
<td>5.5.3</td>
<td>Cache Effectiveness</td>
<td>60</td>
</tr>
<tr>
<td>5.6</td>
<td>Chapter summary</td>
<td>64</td>
</tr>
<tr>
<td>6</td>
<td><strong>Adaptive Wear Balancer for Flash Clusters</strong></td>
<td>65</td>
</tr>
<tr>
<td>6.1</td>
<td>Introduction</td>
<td>65</td>
</tr>
<tr>
<td>6.2</td>
<td>Background and Motivation</td>
<td>68</td>
</tr>
<tr>
<td>6.3</td>
<td>System Design</td>
<td>69</td>
</tr>
<tr>
<td>6.3.1</td>
<td>Overview of Chameleon architecture</td>
<td>70</td>
</tr>
<tr>
<td>6.3.2</td>
<td>Adaptive wear balancing algorithms</td>
<td>71</td>
</tr>
<tr>
<td>6.3.3</td>
<td>Mapping table</td>
<td>75</td>
</tr>
<tr>
<td>6.4</td>
<td>Evaluation</td>
<td>77</td>
</tr>
</tbody>
</table>
6.4.1 Implementation .................................................. 77
6.4.2 Experimental Methodology .................................... 79
6.5 Chapter summary .................................................... 84

7 Conclusion and Future Work ...................................... 86
  7.1 Summary ............................................................ 87
  7.2 Future Directions ................................................... 88
    7.2.1 Low-overhead container network ............................... 88
    7.2.2 High-performance storage system for serverless analytics .... 88
    7.2.3 Tailoring serverless computing for distributed machine learning .... 89

Bibliography ............................................................ 90
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Docker ecosystem</td>
<td>12</td>
</tr>
<tr>
<td>3.2</td>
<td>Crawler, Downloader, and Analyzer</td>
<td>13</td>
</tr>
<tr>
<td>3.3</td>
<td>CDF of layer size</td>
<td>15</td>
</tr>
<tr>
<td>3.4</td>
<td>CDF of image size</td>
<td>15</td>
</tr>
<tr>
<td>3.5</td>
<td>CDF of layer count</td>
<td>15</td>
</tr>
<tr>
<td>3.6</td>
<td>CDF of layer count per image</td>
<td>16</td>
</tr>
<tr>
<td>3.7</td>
<td>Histogram of layer count per image</td>
<td>16</td>
</tr>
<tr>
<td>3.8</td>
<td>CDF of layer reference count</td>
<td>16</td>
</tr>
<tr>
<td>3.9</td>
<td>CDF of file count</td>
<td>17</td>
</tr>
<tr>
<td>3.10</td>
<td>CDF of file size</td>
<td>17</td>
</tr>
<tr>
<td>3.11</td>
<td>CDF of directory count</td>
<td>17</td>
</tr>
<tr>
<td>3.12</td>
<td>CDF of layer directory depth</td>
<td>17</td>
</tr>
<tr>
<td>3.13</td>
<td>CDF of repositories by pull count</td>
<td>17</td>
</tr>
<tr>
<td>3.14</td>
<td>Histogram of repositories by pull count</td>
<td>17</td>
</tr>
<tr>
<td>3.15</td>
<td>Taxonomy of file types</td>
<td>18</td>
</tr>
<tr>
<td>3.16</td>
<td>Common used file types</td>
<td>19</td>
</tr>
<tr>
<td>3.17</td>
<td>EOL files</td>
<td>19</td>
</tr>
<tr>
<td>3.18</td>
<td>Source codes</td>
<td>20</td>
</tr>
<tr>
<td>3.19</td>
<td>File repeat count distribution</td>
<td>21</td>
</tr>
<tr>
<td>3.20</td>
<td>Deduplication ratio growth</td>
<td>21</td>
</tr>
<tr>
<td>3.21</td>
<td>Layer redundant ratio</td>
<td>22</td>
</tr>
</tbody>
</table>
3.22 Image redundant ratio .......................................................... 23
3.23 Overall deduplication ratio .............................................. 24
3.24 Deduplication results for EOL files ................................... 24
3.25 Deduplication results for source codes .............................. 24

4.1 Average pulling/pushing latency breakdown. X-axis shows the layer size percentiles. Y-axis is log-scaled. ........................................ 29
4.2 Average compression ratios. ............................................. 29
4.3 Storage option impact. ...................................................... 31
4.4 Concurrency impact. X-axis shows the layer size percentile and number of concurrent threads. ...................................................... 31
4.5 I/O size impact on reads. X-axes represents I/O size. .............. 34
4.6 I/O size impact on writes. X-axes represents I/O size. .......... 34
4.7 CoW performance of container storage drivers and backend file systems. ................................................................. 36
4.8 Concurrency. X-axis represent the number of read/write threads. ................................................................. 36
4.9 Off-line file-level deduplication run time. ............................. 39

5.1 CDF of GET layer request count ........................................ 46
5.2 CDF of Client repulling probability .................................... 46
5.3 CDF of GET manifest/layer IAT ......................................... 46
5.4 Duphunter architecture ..................................................... 49
5.5 Layer dedup, replication, and partitioning. ............................ 53
5.6 Parallel streaming layer construction. ................................ 54
5.7 Tiered storage architecture. ............................................. 55
5.8 Cache hit ratio on P-servers with different cache algorithms. ... 58
5.9 99th percentile GET layer latency of P-servers. .................. 59
5.10 Layer restoring latency breakdown (X-axis is log-scale). ........ 60
5.11 Preconstruct cache hit ratio. .......................................... 60
5.12 Performance of D-servers ................................................. 60
5.13 Request wait ratio with different number of clients. ............. 60
5.14 Average wait time with different number of clients (Y-axis is log-scale) . . . 61

6.1 Wear imbalance in a 50-server flash cluster. X-axis shows flash servers sorted by total erasure count. ................................................................. 66

6.2 Chameleon system. ................................................................. 69

6.3 Chameleon’s mapping table. .................................................. 76

6.4 Wear variance. ................................................................. 80

6.5 Flash endurance. ............................................................... 81

6.6 Write latency. ................................................................. 82

6.7 Write amplification. ........................................................... 83

6.8 Data state changes over 85 hours under Chameleon by replaying YCSB-zipf workload. ............................................................... 84
List of Tables

4.1 Docker image distribution. .................................................. 29
4.2 Schemes. ........................................................................ 33

5.1 Dedup. ratio vs GET layer latency increases .......................... 47
5.2 Workload parameters. ......................................................... 57
5.3 Dedup. ratio vs. GET layer latency. ..................................... 58

6.1 Terminology & List of Acronyms. ........................................ 73
6.2 SSD parameters. ................................................................. 78
6.3 Trace characteristics. ......................................................... 79
6.4 Test schemes. ................................................................. 79
Chapter 1

Introduction

Application containerization enables easier runtime management, automatic deployment, and horizontal scaling. As a result, containerization framework such as Docker [6] have seen a remarkable adoption in both modern cloud and on-premise environments. Moreover, the global application container market is expected to reach USD $8.2 billion and register a CAGR of 26.5% by 2025 [13]. Storage is critical for container-based applications. There are several reasons. First, containers are created from images. Docker image storage and distribution services significantly impact container startup performance. Second, container storage drivers largely affect container runtime performance, especially I/O performance. Third, container startup time is crucial for containerized applications. These put challenges for container storage systems.

1.1 Motivation

Container images are stored and distributed via a Docker registry. As the container market continues to expand, Docker registries have to manage a growing number of images and layers. The amount of data Docker registries store is massive. Some conservative estimates show that in spring 2019, Docker Hub alone stored at least 2 million public images totaling roughly 1 PB in size [167, 196]. This is just the tip of the iceberg and the number of private images is significantly higher. Other popular public registries [30, 96, 108, 134], as well as on-prem registry deployments in large organizations, experience a similar surge in the number of images. As a result, a comprehensive and large-scale characterization of the images and layers stored in the Docker registry is needed to find interesting properties regarding storage, layer sharing, and redundancy. These properties have implications on the future design of Docker registry and container storage drivers.

The rise of containers has led to a broad proliferation of container images. The associated storage performance and capacity requirements place high pressure on the infrastructure of
registries. Exploiting the high file redundancy in real-world images is a promising approach to drastically reduce the demanding storage requirements of the growing registries. However, existing deduplication techniques [140] [61] [191] [182] significantly degrade the performance of registries because of the high layer restore overhead.

Distributed storage systems have witnessed tremendous performance boost through high-speed SSDs and deliver scalable, persistent, and reliable storage for container orchestration systems such as Kubernetes [17]. NAND flash-based Solid State Devices (SSDs) offer the desirable features of high performance, energy efficiency, and fast growing capacity. A key obstacle in this context is that the natural unbalance in distributed I/O workloads can result in wear imbalance across the SSDs in a distributed setting. This, in turn can have significant impact on the reliability, performance, and lifetime of the storage deployment. Extant load balancers for storage systems do not consider SSD wear imbalance when placing data, as the main design goal of such balancers is to extract higher performance. Consequently, data migration is the only common technique for tackling wear imbalance, where existing data is moved from highly loaded servers to the least loaded ones. However, extra wear is caused by data migration.

To address the above issues, this dissertation proposes, designs, and implements a series of novel techniques, algorithms, and frameworks that can improve the performance, flexibility, and space-efficacy of storage systems for containerized applications. Specifically, this dissertation adopts three steps: (1) analyzing the Docker image dataset; (2) deriving the design implications; (3) designing a new storage framework. The overarching goal of this dissertation is to improve the flexibility, performance, and space-efficacy of storage systems for modern containerized applications by using data analysis, deduplication, and wear balancing.

The next sections describe the research problems, proposed research methodologies, and evaluation results of each work included in this dissertation.

### 1.1.1 Large-Scale Analysis of the Docker Hub Dataset

Docker containers have become a prominent solution for supporting modern enterprise applications due to the highly desirable features of isolation, low overhead, and efficient packaging of the execution environment. Containers are created from images which are shared between users via a Docker registry. The amount of data Docker registries store is massive; for example, Docker Hub, a popular public registry, stores at least half a million public images.

Registries are growing rapidly. For example, Docker Hub [7], the most widely used registry, stores more than 500,000 public image repositories comprising over 2 million layers and it keeps growing. Over a period from June to September 2017, we observed a linear growth of the number of images in Docker Hub with an average creation rate of 1,241 repositories per day. We expect this trend to continue as containers gain more popularity. This massive image dataset presents challenges to the registry storage infrastructure and so far has remained
largely unexplored.

In this work, we perform the first, comprehensive, large-scale characterization and redundancy analysis of the images and layers stored in the Docker Hub registry.

1.1.2 Design Implications for Container Storage Systems via Performance Analysis

Images and layers are stored and managed by container storage drivers. As the performance of these drivers can largely impact container startup, build, and execution time, we perform a comprehensive analysis on popular storage drivers based on the key insights from our image characterizations.

Data deduplication has proven itself as a highly effective technique for eliminating data redundancy. In spite of being successfully applied to numerous real datasets, deduplication bypassed the promising area of Docker images. Given our findings, we propose Slimmer, a file-level content addressable storage model for the Docker registry. Slimmer unpacks layer tarballs into individual files and deduplicates them. When a Docker client requests a layer, Slimmer dynamically reconstructs the layer from its constituent files. To assess the feasibility of our design, we conduct a simulation-based evaluation of Slimmer. The simulation results show that Slimmer improves the deduplication ratio from $1.8\times$, provided by layer sharing, to $6.9\times$. While Slimmer comes with some overhead caused by the need to decompress and reconstruct layers, we found, for example, that for layers less than 10 MB (around 60% of all layers) the overhead of retrieving a layer is less than 1 s. For larger layers, we propose several optimizations to reduce overhead.

1.1.3 Flexible High-performance Deduplication for Docker Registries

The rise of containers has led to a broad proliferation of container images. The associated storage performance and capacity requirements place high pressure on the infrastructure of container registries that store and serve images. Exploiting the high file redundancy in real-world container images is a promising approach to drastically reduce the demanding storage requirements of the growing registries.

Deduplication is an effective method to reduce capacity demands of intrinsically redundant datasets [156]. However, applying deduplication to a Docker registry is challenging due to two main reasons: 1) layers are stored in the registry as compressed tarballs that do not deduplicate well [132]; and 2) decompressing layers first and storing individual files incurs high reconstruction overhead and slows down image pulls. The slowdowns during image pulls are especially harmful because they contribute directly to the startup times of containers.
Our experiments show that, on average, naive deduplication increases layer pull latencies by up to $98\times$ compared to a registry without deduplication.

In this paper, we propose DupHunter, the first Docker registry that natively supports deduplication. It is designed to increase storage efficiency via layer deduplication while reducing the corresponding layer restoring overhead.

### 1.1.4 Adaptive Wear Balancer for Flash Clusters

NAND flash-based Solid State Devices (SSDs) offer the desirable features of high performance, energy efficiency, and fast growing capacity. Thus, the use of SSDs is increasing in distributed storage systems. A key obstacle in this context is that the natural unbalance in distributed I/O workloads can result in wear imbalance across the SSDs in a distributed setting. This, in turn, can have significant impact on the reliability, performance, and lifetime of the storage deployment. Extant load balancers for storage systems do not consider SSD wear imbalance when placing data, as the main design goal of such balancers is to extract higher performance. Consequently, data migration is the only common technique for tackling wear imbalance, where existing data is moved from highly loaded servers to the least loaded ones.

Moreover, wear imbalance worsens when fault tolerant or data redundancy schemes such as replication (REP) or erasure coding [65, 141, 188] (EC) are applied in a flash cluster. This is mainly because storing extra redundant data generates more writes, which in turn severely impact flash endurance. To solve the problems of multi-server wear imbalance, we propose a practical and efficient global wear balancing technique, Chameleon. Chameleon quickly detects the presence of erasure imbalance in a flash cluster. The goal is to balance the erasure count and improve both lifetime and performance of the flash cluster.

### 1.2 Research Contributions

From the above three aspects, we will demonstrate that we can improve the performance and space-efficacy of storage systems for containerized applications.

Overall, this dissertation proposes innovative systemic and algorithmic approaches to tackle the redundancy and inflexibility of the storage system for containerized applications. In the following, we highlight the specific research contributions that this dissertation make.

**Large-Scale Analysis of the Docker Hub Dataset** In this work, we analyze over 167 TB of uncompressed Docker Hub images, characterize them using multiple metrics and evaluate the potential of filelevel deduplication in Docker Hub. Our analysis helps to make conscious decisions when designing storage for containers in general and Docker
registries in particular. For example, only 3% of the files in images are unique, which means file-level deduplication has a great potential to save storage space for the registry. Our findings can motivate and help improve the design of data reduction, caching, and pulling optimizations for registries.

**Design Implications for Container storage systems via performance analysis** In this work, we perform a comprehensive analysis on popular storage drivers based on the key insights from our image characterizations. First, we analyze the layer pulling latency distribution for the images from the Docker Hub dataset and propose various optimizations for speeding up container startup times. Second, we observe that files are small while layers and images contain many files. For example, 90% and 50% of files are smaller than 26KB and 4KB respectively while 80% of images have more than 15K files, which implies that the small I/O request performance is important for container storage drivers. Therefore, we study the small I/O performance for different popular container storage drivers. Based on our analysis, we derive several design implications for storage drivers, e.g., storage drivers should be optimized for both small reads and small writes, especially the small rewrites to the files in the preceding read-only layers.

**Flexible High-performance Deduplication for Docker Registries** In this work, we propose DupHunter, a new Docker registry architecture, which not only natively deduplicates layers for space savings but also reduces layer restore overhead. DupHunter supports several configurable deduplication modes, which provide different levels of storage efficiency, durability, and performance, to support a range of uses. To mitigate the negative impact of deduplication on the image download times, DupHunter introduces a two-tier storage hierarchy with a novel layer prefetch/preconstruct cache algorithm based on user access patterns. Under real workloads, in the highest data reduction mode, DupHunter reduces storage space by up to $6.9 \times$ compared to the current implementations. In the highest performance mode, DupHunter can reduce the GET layer latency up to $2.8 \times$ compared to the state of the art.

**Adaptive Wear Balancer for Flash Clusters** In this work, we explore an innovative holistic approach, Chameleon, that employs data redundancy techniques such as replication and erasure-coding, coupled with endurance-aware write offloading, to mitigate wear level imbalance in distributed SSD-based storage. Chameleon aims to balance the wear among different flash servers while meeting desirable objectives of: extending life of flash servers; improving I/O performance; and avoiding bottlenecks. Evaluation with a 50 node SSD cluster shows that Chameleon reduces the wear distribution deviation by 81% while improving the write performance by up to 33%.
1.3 Dissertation Organization

The rest of the dissertation is organized as follows. In Chapter 2 we introduce the background technologies and state-of-the-art related work that lay the foundation of the research conducted in this dissertation. Chapter 3 presents a large scale analysis of Docker image dataset. Chapter 4 presents performance analysis of container storage systems and design implications. Chapter 5 introduces a flexible deduplication framework that reduces redundant data while maintains a good performance. Chapter 6 describes a wear balancer for flash-based storage clusters. Chapter 7 concludes and discusses the future directions.
Chapter 2

Background

2.1 Docker Registry Storage Systems

The main purpose of a Docker registry is to store and distribute container images to Docker clients. A registry provides a REST API that allows Docker clients to push images to and pull images from it [87, 88]. Docker registries group images into repositories, each containing versions (tags) of the same image, identified as `<repo-name:tag>`. For each tagged image in a repository, the Docker registry stores a manifest, i.e. a JSON file, which contains the runtime configuration for a container image (e.g., environment variables) and the list of layers that make up the image. A layer is stored as a compressed archival file and identified using a digest (SHA-256), computed over the uncompressed content of the layer. When pulling an image, a Docker client first downloads the manifest and then the referenced layers, which are not already present on the client. When pushing an image, a Docker client first uploads the layers (if not already present in the registry) and then the manifest.

The current Docker registry software is a single-node application with a RESTful API. The registry delegates storage to a backend storage system, ranging from local file systems to distributed object storage systems such as Swift [142] or others [2, 3, 105, 142] through the corresponding storage drivers. To scale the registry, organizations typically deploy a load balancer or proxy in front of several independent registry instances [48]. In this case, client requests are forwarded to the destination registries through a proxy, then served by the registries’ backend storage system. To reduce the communication overhead between the proxy, registry, and backend storage system, Bolt[127] proposes to use a consistent hashing function instead of a proxy, distribute requests to registries, and utilize the local file system on each registry node to store data instead of using a remote distributed object storage system. Multiple layer replicas are stored on Bolt registries for high availability and reliability.

Registry performance is critical to Docker clients. In particular, the layer pulling performance (i.e. GET layer performance) impacts container startup times [102] significantly.
Related work has studied various dimensions of registry performance and a Docker image dataset [48, 64, 102, 170, 177, 192, 196]. However, existing work does not provide deduplication capabilities to the registry. A community proposal exists to add file-level deduplication to container images [29], but as of now lacks a detailed design or performance analysis. Skourtis et al. [167] propose restructuring layers to optimize for various dimensions, including registry storage utilization. Their approach does not remove all duplicates. Much work aims to reduce the size of a single image [89, 99, 159, 179] to reduce network latency.

### 2.2 Deduplication in Storage Systems

Data deduplication has received considerable attention, particularly for virtual machine images [106, 111, 171, 199]. Many deduplication studies focus on primary and backup data deduplication [91, 92, 93, 125, 126, 128, 138, 163, 178, 184, 200] and show the effectiveness of file- and block-level deduplication [131, 173]. To further reduce storage space, block-level deduplication integrating with compression is proposed [181]. In addition to local deduplication schemes, a global deduplication method [140] is proposed to improve the deduplication ratio and provide high scalability for distributed storage systems.

Data restoring latency is an important factor for storage systems with deduplication support. Efficient chunk caching algorithms and forward assembly are proposed to accelerate data restore performance [66].

### 2.3 SSD-based storage cluster

**Flash endurance**  A large body of work has examined flash endurance [56, 73, 110, 169]. Techniques such as log-structured caching [169], inclusion of combining multiple bad blocks into virtual healthy blocks [110] have been explored to improve the lifetime of flash devices. These works are orthogonal and complementary to Chameleon.

**Intradisk wear leveling**  Dynamic [113, 129] techniques aim to achieve a good wear evenness while keeping the overhead low. Similarly, static wear leveling techniques [71, 109, 110, 137] move cold data to the blocks with higher erasure counts, thereby improving the even spread of wear. Chameleon leverages such approaches for extending the lifetime of individual SSDs in its target distributed setting.

**Interdisk wear leveling**  Application of SSD arrays in enterprise data-intensive applications is growing. In such an environment, we have observed significant variance in number of writes and merge operations that are sent to individual SSDs. Recent work [98] manages
2.3. SSD-based storage cluster

EC stripes to increase reliability and operational lifetime of such flash memory-based storage systems, and uses a log-structured approach that does not need explicit wear balancing as data is appended and not updated in place. In contrast, EDM [145] also targets SSD arrays but use data migration to achieve wear balance across the SSDs in the array. SWANS [186] dynamically monitors the variance of write intensity across the array and redistributes writes based only on the number of writes that an SSD has received to prolong the SSD arrays’ service life. These methods share with Chameleon the goal of wear leveling across an SSD array, however unlike them Chameleon considers the role of redundant policies at various storage hierarchy and their impact on overall wear balancing.

Distributed flash storage systems  FAWN [32] uses small amounts of local flash storage across a number of low-power resource-constrained nodes to enable a consistent and replicated key-value storage system. CORFU [52] extends the local log-structured design by organizing the entire cluster of SSDs as a global shared log. Both of these systems utilize homogeneous nodes and replication for high availability. Other works [35, 76, 77, 120] focuses on tiered storage to reduce the load on flash devices. Similarly, [34, 36] use data partitioning to evenly distribute load. In contrast, Chameleon focuses on EC storage solutions, which offer higher storage efficiency and exploits the interactions between the storage hierarchy to improve overall flash lifetime in flash-based clusters.
Chapter 3

Large-Scale Analysis of Docker Images

3.1 Introduction

Recently, containers [133] have gained significant traction as an alternative to virtual machines [162] for virtualization both on premises and in the cloud. Polls suggest that 87% of enterprises are in the process of adopting containers, and that containers are expected to constitute a lucrative $2.5 billion market by 2020 [26]. In contrast to Virtual Machines (VMs), containers share the same kernel but are isolated in terms of process visibility (e.g., via namespaces [19]) and resource usage (e.g., via control groups [16]). Containers require fewer memory and storage resources, start faster, and typically incur less execution overhead than VMs [62, 90, 136].

A driving force for fast container adoption is the popular Docker [6] container management framework. Docker combines process containerization with convenient packaging of an application’s complete runtime environment in images. For storage and network efficiency, images are composed of independent, shareable layers of files. Images and their corresponding layers are stored in a centralized registry and accessed by clients as needed. Docker Hub [7] is the most popular registry, currently storing more than 500,000 public image repositories comprising over 2 million layers. The size of the registry is steadily increasing. Over a period from June to September 2017, we observed a linear growth of the number of images in Docker Hub with an average creation rate of 1,241 public repositories per day. We expect this trend to continue as containers gain more popularity.

While the massive image dataset presents challenges to the registry and client storage infrastructure, storage for containers has remained a largely unexplored area. We believe one of the prime reasons is the limited understanding of what data is stored inside containers. This knowledge can help improve the container storage infrastructure and ensure scalability of and fast accesses to the registry service. Existing work has focused on various aspects of containerization [54, 60, 79, 102, 164, 197]. However, the registry and its contents have yet
3.2 Background and Methodology

3.2.1 Background

Container-based virtualization (such as Linux Containers (LXC) [1]) has emerged as a lightweight virtualization alternative. Compared to Virtual Machine based server virtualization technologies [84, 152, 153, 154] (e.g., VMware [172] or Xen [53]), container virtualization works at the operating system level. Containers share the same kernel which improves startup times and significantly reduces the storage and memory overhead [168].

Docker Docker is a popular containerization framework, which extends LXC with higher level APIs and additional functionality [27, 82, 151]. It automates the deployment of applications inside Linux containers, and provides the capability to package an application with its runtime dependencies into a container [102].

Our analysis reveals several interesting insights. First, the majority of layers are small in size and show a low compression ratio. 50% of the layers are smaller than 4 MB which holds both for compressed and uncompressed layers, and the median layer compression ratio is 2.6. As compression is computationally intensive, storing small layers in the registry uncompressed can improve latency during pulls as layers do not have to be uncompressed locally anymore. Second, we find that only around 3% of the files are unique while others are redundant copies. This suggests that file-level deduplication has a great potential to save storage space for large-scale registries.

We also find that image accesses are skewed towards a small number of popular images. Specifically, 90% of repositories are pulled less than 300 times since creation, while the largest number of pulls we record for an image is over 600 million. This suggests that image caching is a viable improvement for the registry. Our analysis provides a first insight into the Docker image dataset, which can help improve the design of current data reduction, caching, and pulling optimizations for container registries.
As shown in Figure 3.1, the Docker ecosystem consists of several components. Users interact with Docker via the Docker client, which sends commands to the Docker daemon. The daemon is responsible for running containers from locally available images. Additionally, the daemon supports building new images and pushing them to a Docker registry. When a user wants to launch a container from an image that is not available locally, the daemon pulls the required image from the registry.

**Docker images and layers** At the center of Docker is the concept of container images for packaging, distributing, and running applications. Docker images consist of a series of individual layers. A layer contains a subset of the files in the image and often represents a specific component/dependency of the image, e.g., a library. This modular design allows layers to be shared between two images if both images depend on the same component.

Image layers are read-only. When users start a container, Docker creates a new writable layer on top of the underlying read-only layers as shown in Figure 3.1. Any changes made to files in the image will be reflected inside the writable layer via a copy-on-write mechanism. This leaves image layers unmodified throughout the lifetime of a container and enables layer sharing. Docker supports multiple storage drivers, e.g., Aufs and Btrfs, which efficiently combine read-only and writable layers in a single file-system namespace and support copy-on-write [177]. The writable layers are discarded when the container is deleted.

An image is represented by a manifest file, which contains a list of layer identifiers (digests) for all layers required by the image. Moreover, it describes the various parameters of a Docker image, such as the target hardware platform and environment settings.

**Docker registry** The Docker registry is a platform for storing and sharing container images. It stores images in repositories, each containing different versions of the same image. Image layers are stored as compressed archival files and image manifests as JSON-based files.
3.2. Background and Methodology

Docker Hub is one of the most popular public registries, supporting both public and private repositories, where users can upload, search, and download images [7]. In Docker Hub, the user repositories are namespaced by user name, i.e., <username>/<repository name>, while the official repositories, which are directly provided by Docker Inc. and partners, are called <repository name>.

3.2.2 Methodology

Our image analysis methodology consists of three steps (see Figure 3.2): i) crawl Docker Hub to list all repositories; ii) download the latest version of an image and all referenced layers from each repository based on the crawler results; iii) decompress and analyze images and layers.

Crawler To download a particular image, the name of the repository which the image belongs to needs to be provided. The crawler is responsible for generating a list of repositories for the downloader.

Public repositories in Docker Hub (i.e., the repositories that anyone can pull from) are divided into official repositories, served by the Docker Hub partners, and non-official repositories, provided by regular users and third-party organizations. The number of official repositories is less than 200, while, the majority of repositories in Docker Hub are non-official (over 400,000). Listing non-official repositories requires web crawling because Docker Hub does not support an API to retrieve all repository names.

Our crawler utilizes the Docker Hub’s Web-based search engine to find all available repositories. As the name of non-official repositories is comprised of the user name and the repository name separated by a “/”, we can search for “/” and obtain a list of all non-official reposi-
Chapter 3. Large-Scale Analysis of Docker Images

The Crawler downloads all pages from the search results and parses the web content to build a list of all non-official repositories. We ran the crawler on May 30th, 2017 and it delivered a list of 634,412 repositories. After removing duplicate entries (introduced by Docker Hub indexing logic), the final repository list consists of 457,627 distinct repositories.

Downloader

Images in Docker Hub repositories are labeled with version tags to track different image versions. If a user does not provide a tag when pulling an image, Docker client pulls the latest tag by default. In this work we focus on downloading images with the latest tag to make the analysis more feasible. We plan to extend our analysis to other image tags in the future.

Instead of using the Docker client to download images, we implement our own downloader, which calls the Docker registry API directly [10] to download manifests and image layers in parallel. Note that we only download unique layers. Our downloader runs significantly faster than a docker pull-based downloader which performs many other operations in addition to downloading the image. For example, it automatically extracts each layer’s tar archive file and creates the corresponding read-only snapshot using the configured Docker storage driver. This not only takes considerable amount of time but also leads to overly high storage space utilization. Furthermore, the local storage format of Docker images makes it difficult to analyze the contents of each layer separately. Our downloader can download multiple images simultaneously and fetch the individual layers of an image in parallel. Layers are transferred as gzip compressed tar archives.

The whole downloading process took around 30 days. Overall, we downloaded 355,319 images, resulting in 1,792,609 compressed layers and 5,278,465,130 files, with a total compressed dataset size of 47 TB. A total of 111,384 images could not be downloaded due to two reasons: 1) 13% of these images required authentication; 2) 87% of these images did not have a latest tag.

Analyzer

The analyzer extracts the downloaded layers and analyzes them along with the image manifests. For each image, it creates an image profile and individual layer profiles, which contain metrics for the whole image and its individual layers, respectively.

To produce the layer profile, the analyzer first decompresses and extracts each layer tarball to a layer directory. Then, it recursively traverses each subdirectory and obtains its metadata information. To create the image profile, the analyzer parses the manifest and obtains its metadata information such as OS and target architecture. Further, once individual layers are analyzed, the analyzer builds the image profile by including pointers to its layer profiles.
3.3. Dataset Characterization

In this section we present our analysis of the Docker Hub dataset by characterizing layers, images, and files.

3.3.1 Layers and images

We start by analyzing layers and images in terms of size and compressibility, file and directory counts, and directory depths.

Layer and image sizes  We characterize layer sizes using two different metrics: 1) compressed layer size — the format a layer is stored in the registry or transferred to a client; and 2) uncompressed layer size — the sum of the sizes of the uncompressed files contained in the layer. Figure 3.3 shows the CDF of the two metrics.

We see that 90% of the layers are smaller than 177 MB in uncompressed format and smaller than 63 MB in compressed format. Interestingly, about half of the layers are smaller than 4 MB, independent of the format.

Similarly to layers, we also measure compressed image size, i.e. the sum of the sizes of the compressed image layers, and the sum of the sizes of files contained in the image as shown in Figure 3.4. 90% of the images have an uncompressed size less than 1.3 GB while compressed images are less than 0.48 GB. In the median, this decreases to 94 MB and 17 MB, respectively. The largest uncompressed image is 498 GB which is a Ubuntu-based image. Figure 3.4 shows that the majority of uncompressed images in Docker Hub are small which aligns with the Docker philosophy to package software and distribute software in containers but include only its necessary dependencies.

Figure 3.3: CDF of layer size.

Figure 3.4: CDF of image size.

Figure 3.5: CDF of layer count.
Chapter 3. Large-Scale Analysis of Docker Images

Figure 3.6: CDF of layer count per image.  
Figure 3.7: Histogram of layer count per image.  
Figure 3.8: CDF of layer reference count.

Compression ratio distribution  To further study the sizes and the impact of compression, we calculate the compression ratios (see Figure 3.5) by using the uncompressed layer archival size divided by compressed layer size. 90% of layers have a compression ratio less than 29. The median layer compression ratio is 3.5 while the largest compression ratio is 1026. Images have a smaller compression ratio than layers. For example, the median image compression ratio is 3.2. Figure 3.5 suggests that layer archivals have a great potential for compression to reduce layer transfer latency.

Layer counts per image  Images consist of a set of layers. It is important to understand the layer count of the images as previous work found that the number of layers can impact the performance of I/O operations [102]. Therefore, we count the number of layers per image (see Figure 3.6) and layer count frequencies (see Figure 3.7) for all Docker Hub images.

As shown in Figure 3.6 that 90% of the images have less than 18 layers while half of the images have less than 8 layers. 8 layers is also the most frequent layer count per image with 51,300 images consisting of exactly 8 layers as shown in Figure 3.7. The maximum layer count is 120 in the \textit{cfgarden/120-layer-image}. We also find that there are 7,060 images that consist of only a single layer.

Layer reference count distribution  Docker registry uses layer-level content addressable to remove layer duplicates. Therefore, layers can be shared among different images. Figure 3.8 shows that around 90% of layers are only reference by a single image while 95% are reference by not more than 2 images. 99% of layers are shared among less than 25 images.

File counts and file sizes  Figure 3.9 shows that 90% of layers contain less than 7,410 files while half of the layers have less than 30 files. We also found that 27% of the layers only have a single file while 7% even showed no files at all. On the other hand, the largest layer contains 826,196 files and was part of a Debian image.
Images have more files than layers. As shown in Figure 3.9, 90% of images have less than 64,780 files. Half of images have less than 1,090 files.

Figure 3.10 shows that files are small. 90% of files are smaller than 26 KB.

**Directory counts and directory depths** Figures 3.11 shows the directory counts in layers and images, respectively. For directories, 90% of the layers have less than 826 directories and half of the layers consist of less than 11 directories. We observe a wide range with a minimum of a single directory and a maximum of 111,940. The layer with the most directories was part of the conjurinc/developer-quiz image.

Besides the count, we also calculate the maximum directory depth for each layer (Figure 3.12). Around 90% of all layers have a directory depth less than 10 while for 50% of the layers, the directory depth is less than 4. The most frequent directory depth is 3 with 313,000 layers showing this depth value.

This analysis shows that the majority of layers consist only of a small number of files and does not contain deeply nested directory hierarchies. Hence, except for few outliers, unpacked layers do not require a large amount of metadata from the storage system.
3.3.2 Files

After analyzing layers and images, we conducted a deeper analysis on the files that are stored in containers. Specifically, we characterize files in terms of size and type. Based on this characterization, we create a three-level classification hierarchy as shown in Figure 3.15. At the highest level, we created two categories: Commonly used file types and non-commonly used file types based on the total file size and file count for each type. Totally, we got around 1,500 types after analyzing our whole dataset. We found that only 133 file types take up more than 7 GB individually and occupy the most capacity (98.4%, with 166.8 TB) totally. We put these 133 file types into commonly used file type group and the remaining files into non-commonly used file types. Our further classification expands on the 98.4% commonly used file types.

At the second level of the hierarchy, we clustered commonly used file types based on the
3.3. Dataset Characterization

(a) File count by file type group. (b) Capacity by file type group. (c) Average file size by file type group.

Figure 3.16: Common used file types

(a) File count by file type. (b) Capacity by file type.

Figure 3.17: EOL files

major file format, usage, or platform involved by each file type. We identified commonly used file types relevant to EOL (executable, object code, and libraries), source code, scripts, documents, archival, images, databases, and others.

At the third level, we present the specific file types which take a large percentage of storage space.

**Common used file types** Figure 3.16 shows the 8 type groups in terms of file count and capacity. 13%, 11%, and 9% of files are source code, EOL, and scripts. EOL files occupy the most capacity (37%).

We also see that 44% of files are document files such as Microsoft office files, LaTeX files, etc. Only 4% of files are image data files, e.g., PNG, JPEG, etc. Besides, we found a small amount of video files like AVI, MPEG, etc.

To find how file type relate to file size, we calculated the average file size by file type group as shown in Figure 3.16(c). We see that Database files are much bigger (978.8 KB) than the files within other type groups. The average size of EOL and Archival files are around 100 KB.
Chapter 3. Large-Scale Analysis of Docker Images

Executable, object code, and libraries (EOL) Based on the third-level classification, we further investigate the file size and file count by specific file types. We start with EOL group which contains the following types: ELF files, COFF files, intermediate representation that can be executed by a virtual machine, Microsoft executables, Debian/RPM binary packages, libraries, and other EOL files.

Figure 3.17 shows file type distribution in terms of file count and capacity for EOL type group. We see that majority of EOL files are ELF and intermediate representations (shown as “Com.” in the Figures). ELF files mainly contain ELF relocatables, shared objects, and executables. Intermediate representations mainly contain Python byte-compiled files (majority), compiled java class, and terminfo compiled files. Although intermediate representations take up to 64% of EOL files, 30% of EOL files occupy 84% of storage space consumed by the EOL group. This is because average ELF file size is 312 KB while the average intermediate compiled file size is 9 KB. In addition to ELF files and representations, we found Microsoft executables (2%) and Mach-O files (<0.01%) in the EOL group.

We conclude that among all file types, ELF files occupy most capacity. There are large amount of intermediate representations but they take much less storage space.

To know what kind of libraries are used in Docker containers, we categorized the libraries. The most popular libraries we found are: the Palm OS dynamic library, the OCaml library, and the GNU C library.

Source code (SC.) Next, we inspect what kind of languages are commonly used by Docker developers. Figure 3.18 shows 7 major types of source codes in our dataset: C/C++, Perl5 module, Ruby module, Pascal, Fortran, Applesoft basic, and Lisp/Scheme. 80.3% of source files are C/C++ sources which take about 80% of storage space within the source code group. Perl5 module source code and Ruby module source code have an almost similar percentage in terms of file count (9% for Perl5 module source and 8% for Ruby module source) but occupy different percentage in terms of capacity (11% for Perl5 modules and 3% for Ruby modules).
3.4 Deduplication Analysis

In this section, we investigate the potential for data reduction in the Docker registry by analyzing the efficacy of layer sharing and file-level deduplication.

3.4.1 File level deduplication

Next, we calculate the deduplication ratio in terms of file count and capacity for the complete dataset. After removing redundant files, there are only 3.2% of files left, which in total occupy 24 TB, resulting in deduplication ratios of $31.5 \times$ and $6.9 \times$ in terms of file count and capacity, respectively.

We further analyze the repeat count for every file (see Figure 3.19). We observe that over 99.4% of files have more than one copy. Around 50% of files have exactly 4 copies and 90% of files have 10 or less copies. The file that has the maximum repeat count of 53,654,306 is an empty file. Around 4% of empty files are \_init\_py files, which make Python treat a directory as containing packages and are usually empty. Other frequent empty files include \lock\ or \.gitkeep\ files.

We also analyze the top five most frequently repeated files which repeat between 3,338,145 and 11,847,356 times. Specifically, two files \libkrb5-3:amd64.potrm\ and \libkrb5-3:amd64.postinst\ are two Kerberos runtime libraries for \dpkg\. Another two files are related to the \npm\ package manager (\license\ and \.npmignore\) and the last file, \dependency_links.txt\, contains a list of dependency URLs for Python.

This shows that there is a high file-level redundancy in Docker images which cannot be addressed by the existing layer sharing mechanism. Hence, there is a large potential for file deduplication in the Docker registry.
Chapter 3. Large-Scale Analysis of Docker Images

Deduplication ratio growth  To further study the potential of file-level deduplication, we analyze the deduplication for an increasing number of files stored in the registry (see Figure 3.20). The x-axis values correspond to the sizes of 4 random samples drawn from the whole dataset and the size of the whole dataset.

We see that the deduplication ratio increases almost linearly with the layer dataset size. In terms of file count, it increases from $3.6 \times$ to $31.5 \times$ while in terms of capacity, it increases from $1.9 \times$ to $6.9 \times$ as the layer dataset grows from 1000 to 1.7 million layers. This confirms the high potential for file-level deduplication in large-scale Docker registry deployments.

Cross layer file duplicates  Based on the high deduplication ratio, we conclude that a large amount of files are shared between layers and the large potential from deduplication is due to large amount of file duplicates among layers. Cross layer file duplicates are files that are stored in more than one layer, which cannot be eliminated by the layer-level-sharing mechanism. This could be a common problem for layers in the Docker registry. For example, different developers may use the same libraries and build same executables in their layers. But only few files in their layers are different from each other, which makes their layers different.

Figure 3.21 shows the percentage of cross layer file duplicates for each layer. We find that 90% of layers contain more than 97.6% of files that are duplicated across layers. We also calculate the percentage of files that are duplicated across images. As shown in Figure 3.22, 90% of images contain more than 99.4% of files that are duplicated across images, indicating that majority of files are duplicated across different images and layers.
3.4. Deduplication Analysis

(a) CDF of intra/inter-image redundant ratio in terms of file count (denoted as *-cnt.) / capacity (denoted as *-cap.)

(b) Histogram of intra/inter-image redundant ratio in terms of file count (denoted as *-cnt.) / capacity (denoted as *-cap.)

Figure 3.22: Image redundant ratio

3.4.2 Deduplication by file types

To understand what are the file duplicates and why there are so many file duplicates, we look at the deduplication results from the perspective of file types. In this section, we present the deduplication results for common file types that occupy the most capacity.

Figure 3.23 shows deduplication results for the following type groups: EOL, archival, documents, source code, scripts, images, and databases. Note that the y-axes show the capacity occupied by different type groups and their deduplication ratios.

The overall deduplication ratio is 85.69%, and most of the type groups have a comparable ratio. For example, 86% of EOL files, which include executables, object files, and libraries, can be deduplicated at file-level. Source codes, scripts, and documents have the highest deduplication ratio (96.8% for source codes, 98% for scripts, and 92% for documents), which means that Docker developers are more prone to duplicate source code, scripts, and documents.

Next, we see that EOL files, archival, and images have a similar deduplication ratio of around 86%. Compared to other type groups, the redundant EOL files and archival files occupy over half of the capacity (51.4%). Database related files have the lowest deduplication ratio (76%), which contributes little to the overall savings.

Executable, object code, and libraries (EOL) We further calculate the deduplication ratio for specific file types in each common type group. We start from the EOL group since it occupies the most capacity and contributes a lot to the overall savings after deduplication.

Figure 3.24 shows the deduplication results for EOL files. We see that ELF files, intermediate representations, and PE files have the highest deduplication ratio (around 87%). Especially, the redundant ELF files occupy the most capacity (73.4%). Libraries and COFF files have
the lowest deduplication ratio of 53.5% and 61% respectively.

We also calculate the deduplication ratio for each intermediate representation and libraries. We found that all the intermediate representations have a high deduplication ratio (greater than 77%). Especially, the redundant Python byte-compiled code take up to 67% of capacity occupied by intermediate representations. Although the overall deduplication ratio of the library group is lower, we observed that the GNU C/C++ library and the Palm OS dynamic library have a deduplication ratio of over 90%.

**Source code (SC.)** As discussed, Docker developers are more prone to replicate source code. To find out which kind of source codes are replicated frequently, we study deduplication on 7 common languages as shown in Figure 3.25.

We see that all the languages have a high deduplication ratio of over 90% except for Lisp/Scheme. In particular, redundant C/C++ source files take up over 77% of capacity occupied by source code files. To find out why there are so many duplicate C/C++ source files, we inspect those files and find a frequently reused sources related to Google Test [14], a cross-platform C++ test framework available on GitHub [14]. Interestingly, we also observe that there are a large number of repositories related to Google Test but there is no official repository. We suspect that many developers replicate open source code from external public repositories, such as GitHub, and store it in their container images. This could also explain why there are so many shared source code files across different images. Considering that Docker Hub allows developer to automatically build images from source code in external public repositories and automatically push the built image to their Docker
repositories, we believe that replicated source code in different images is a common case in the Docker Hub registry.

3.5 Chapter summary

In this chapter, we carried out the first comprehensive analysis of container images stored in Docker Hub. We presented a methodology to exhaustively crawl and efficiently download Docker Hub images. Using this approach, we analyzed a 47 TB dataset resulting in 1,792,609 layers and 5,278,465,130 files. Based on this dataset, we carried out a detailed study of a variety of storage metrics on both layers, images, and files. Metrics included layer and image sizes, compressibility, deduplication ratio, and popularity. Our findings reveal that there is room for optimizing how images are stored and used. For example, we observed that compression may not always be beneficial for small layers as it can increase pull latencies. Additionally, layers are rarely shared between images which increases storage utilization. Moreover, file-level deduplication can eliminate 96.8% of the files. We plan to investigate such improvements in the future.
Chapter 4

Design Implications for Container Storage Systems via Performance Analysis

4.1 Introduction

Docker containers have become a prominent solution for supporting modern enterprise applications due to the highly desirable features of isolation, low overhead, and efficient packaging of the execution environment. Containers are created from images which are shared between users via a Docker registry. The amount of data Docker registries store is massive; for example, Docker Hub, a popular public registry, stores at least half a million public images. In Chapter 3, we analyze over 167 TB of uncompressed Docker Hub images, characterize them using multiple metrics and evaluate the potential of file-level deduplication in Docker Hub. Our analysis helps to make conscious decisions when designing storage for containers in general and Docker registries in particular. For example, only 3% of the files in images are unique while others are redundant file copies, which means file-level deduplication has a great potential to save storage space.

Images and layers are stored and managed by container storage drivers. As the performance of these drivers can largely impact container startup, build, and execution time, we perform a comprehensive analysis on popular storage drivers based on the key insights from our image characterizations. First, we analyze the layer pulling latency distribution for the images from the Docker Hub dataset and propose various optimizations for speeding up container startup times. Second, we observe that files are small while layers and images contain many files. For example, 90% and 50% of files are smaller than 26 KB and 4 KB respectively while 80% of images have more than 15 K files, which implies that the small I/O request performance is important for container storage drivers. Therefore, we study the small I/O
performance for different popular container storage drivers. Based on our analysis, we derive several design implications for storage drivers, e.g., storage drivers should be optimized for both small reads and small writes, especially the small rewrites to the files in the preceding read-only layers.

4.2 Background

4.2.1 Container storage drivers

Docker employs storage drivers to manage layers and provide the container root file system. The root file system is a unification of the image layers under a single mount point, achieved through techniques such as union file systems or snapshots. For example, Docker’s recommended driver is the overlay2 driver, which uses OverlayFS to unify layers [8]. Docker can retrieve layers via the network and write layers to local disk via storage driver. If the layer downloading process fails prior to the layer being written to disk, Docker can simply re-download the layer. In this case, storage driver can avoid performing costly operations such as fsync, journaling, and check-pointing.

Docker supports multiple storage drivers, e.g., overlay2 [22], devicemapper [5], zfs [191], and btrfs [61], which efficiently manage read-only and writable layers in a single file-system namespace using a copy-on-write (CoW) approach [177]. According to the copy-on-write granularity of different storage drivers, we split the storage drivers into two main groups:

**File-level CoW.** Union file system-based drivers such as overlay2 employ a file-level CoW technique. When an upper layer writes to a file that already exists in a lower layer, the storage driver creates a new copy of the file and applies the modification to the copy. The file in the lower layer is not modified because other containers may refer to the same layer. This approach incurs a performance and disk usage penalty as it copies the entire file from a lower layer when the container modifies the file, even if the modification is small. Moreover, union file system also causes inode proliferation.

**Block-level CoW.** Btrfs and devicemapper are block-level CoW storage drivers. Compared to the file-level drivers above, block-level drivers perform CoW at block granularity, i.e. instead of copying an entire file when modified, only the modified block(s) are copied. Consequently, these drivers provide more efficient usage of disk space and incur less overhead during writes.

4.2.2 Related work on storage drivers.

The container storage driver is one of the key components for efficient execution of a container [24, 100, 101, 119, 158, 175]. Hence, it is important to understand the performance
implications of storage drivers. A number of research papers analyzed the performance of container storage drivers. Xu et al. presented a characterization of the performance impact among various storage and driver options and provided configuration options for better utilizing high speed SSDs [189]. Bhimani et al. [54] characterized the performance of persistent storage options for I/O intensive containerized applications with NVMe SSDs. Tarasov et al. discussed the influence of different storage solutions on different IO workloads, running inside containers [176].

However, these studies are not based on a real Docker image dataset and did not consider the content and storage properties of real images. For instance, we observe that the majority of files in layers are small while images and layers have many files. But the above researchers mostly measured the I/O performance by reading/writing a few big files, or using bigger I/O requests. To analyze the performance of storage drivers, we first explore the image, layer, and file distributions. After that, we perform a comprehensive analysis of popular storage drivers based on real image characteristics in Section 4.3 and Section 4.4.

4.3 Layer Pulling Latency Analysis

Layer pulling performance is critical as it largely affects the container startup time. In this section, we measure the layer pulling latency distribution, identify the performance bottleneck during layer pulling, and propose different methods to reduce the latency. In particular, we aim to address the following questions.

1. What is the overall layer pull latency distribution for Docker registries?
2. What is the bottleneck during layer pulling?
3. How do different compression methods impact layer pulling latency?
4. How do different storage options affect layer pulling latency?

Testbed. Our testbed consists of two servers, one running a Docker client and the other running a Docker registry. Each server is equipped with 32 cores, 64 GB RAM, a 500 GB SSD, a 1 TB HDD, and a 10 Gbps NIC.

Layer dataset To measure the overall layer pulling performance distribution, we first group the layer dataset by layer size. To capture the overall layer pulling performance distribution, we select five groups at the 20th, 40th, 60th, 80th, and 99th percentile, according to layer size distribution shown in Table 4.1. Then, we randomly sample 1000 layers from each group for layer pulling performance experiments.
4.3. Layer Pulling Latency Analysis

Table 4.1: Docker image distribution.

<table>
<thead>
<tr>
<th></th>
<th>20\textsuperscript{th} percentile</th>
<th>40\textsuperscript{th} percentile</th>
<th>60\textsuperscript{th} percentile</th>
<th>80\textsuperscript{th} percentile</th>
<th>99\textsuperscript{th} percentile</th>
</tr>
</thead>
<tbody>
<tr>
<td>File size</td>
<td>1 KB</td>
<td>2 KB</td>
<td>6KB</td>
<td>18 KB</td>
<td>1 MB</td>
</tr>
<tr>
<td>File cnt. / layer</td>
<td>1</td>
<td>2</td>
<td>90</td>
<td>2.6K</td>
<td>50K</td>
</tr>
<tr>
<td>File cnt. / Img.</td>
<td>15 K</td>
<td>20 K</td>
<td>35K</td>
<td>50K</td>
<td>230K</td>
</tr>
<tr>
<td>Layer cnt. / Img.</td>
<td>6</td>
<td>10</td>
<td>12</td>
<td>19</td>
<td>50</td>
</tr>
<tr>
<td>Dir depth / layer</td>
<td>1</td>
<td>3</td>
<td>5</td>
<td>8</td>
<td>18</td>
</tr>
<tr>
<td>Dir cnt. / layer</td>
<td>3</td>
<td>4</td>
<td>34</td>
<td>400</td>
<td>7.5K</td>
</tr>
<tr>
<td>Dir cnt. / Img.</td>
<td>1.8K</td>
<td>3K</td>
<td>4.2K</td>
<td>6.6K</td>
<td>33K</td>
</tr>
<tr>
<td>Layer size</td>
<td>3 KB</td>
<td>14 KB</td>
<td>7.8 MB</td>
<td>53.7 MB</td>
<td>878.9 MB</td>
</tr>
<tr>
<td>Img. size</td>
<td>190 MB</td>
<td>280 MB</td>
<td>530 MB</td>
<td>800 MB</td>
<td>4.9 GB</td>
</tr>
</tbody>
</table>

Figure 4.1: Average pulling/pushing latency breakdown. X-axis shows the layer size percentiles. Y-axis is log-scaled.

4.3.1 Performance breakdown

Pulling a layer includes layer transfer, layer decompression, and layer unpacking while layer pushing includes layer packing, layer compression, and layer transfer. Layer compression can significantly reduce layer size and network transfer time. We first study the impact of the following three popular compression methods on layer pulling and pushing latency: (1) Gzip, the default compression method for Docker. (2) Pigz, a parallel gzip compression method, and (3) Lz4, a fast byte-oriented compression scheme. Note that we choose a compression level of 9 for all three methods, which is the high compression mode.

Figure 4.1(a) shows the average layer pulling latency breakdown for layers at different size
percentiles with different compression algorithms. In this experiment, the Docker client stores layers and their uncompressed content on a tmpfs disk to save packing and unpacking time. As shown in Figure 4.1(a), the layer pulling latency increases dramatically with layer size. Layers at the 20\textsuperscript{th} (3 KB) and 40\textsuperscript{th} percentile (14 KB) show a similar layer pulling latency of 0.03 s for all the three compression algorithms because layers are small. When the layer size increases from 7.8 MB (60\textsuperscript{th} percentile) to 878.9 MB (99\textsuperscript{th} percentile), layer pulling latency increases from 0.36 s to 29 s for Gzip.

Among the three compression algorithms, Gzip has the lowest decompression speed, especially for large layers. For example, it takes 18.7 s for Gzip to decompress a 878.9 MB layer (99\textsuperscript{th} percentile) on average. While for Pigz and LZ4, decompression time is decreased to 10.6 s and 6 s, respectively. Although LZ4 is up to 3.1× and 1.8× faster than Gzip and Pigz, respectively, we observe that the network transfer time for LZ4 is slightly higher (1.2×) than both Gzip and Pigz. This is because the average compression ratio for LZ4 is slightly lower (1.2×) compared to Gzip and Pigz as shown in Figure 4.2. Consequently, there is a trade-off between compression ratio (network transfer time) and compression/decompression speed.

Both Gzip and Pigz show a similar compression ratio because both are based on the DEFLATE algorithm [143] and Pigz only parallelizes Gzip compression to achieve a faster compression. As shown in Figure 4.2, we also observe that the compression ratios decrease with layer sizes. For example, when a layer size increases from 3 KB (20\textsuperscript{th} percentile) to 898.9 MB (99\textsuperscript{th} percentile), the compression ratio decreases from 25.3 to 2.3 for both Gzip and Pigz.

As shown in Figure 4.1(a), when a tmpfs file system is used for storing unpacked layer contents, the network transfer time is the bottleneck for layer pulling performance for 40% of the layers (layers smaller than 14 KB). For larger layers, decompression time becomes the bottleneck. Although network transfer time also increases with layer size, decompression time increases much faster. This is because, in our setup, the Docker registry and Docker client are located in the same fast LAN. In this case, a fast compression algorithm, such as LZ4 or Pigz, is crucial for speeding up layer pulling. However, if images are accessed from a remote registry over a wide area network, the bottleneck can shift again back to the network.

Figure 4.1(b) shows the layer pushing latency distribution. We observe that push latencies are much higher than pull latencies. This is because compression is more expensive than decompression. For example, it takes 18 s for Gzip to decompress a 878.9 MB layer while it takes 380.2 s to compress it. LZ4 has the fastest compression compared to the other two algorithms and is 43.2× and 2.3× faster than Gzip and Pigz, respectively, when layers are at 99\textsuperscript{th} percentile (878.9 MB). Consequently, a fast compression algorithm, such as LZ4 or Pigz, is important for improving both layer pulling and pushing performance.
4.3. Layer Pulling Latency Analysis

Next, we investigate if packing/unpacking can become a bottleneck for layer pushing/pulling. Figure 4.3 shows the average packing and unpacking latency distribution by using different storage media to store the uncompressed and unpacked layer content. As expected, unpacking takes much longer than packing because unpacking involves more writes. For example, when an HDD is used to store the unpacked layer content, it takes 53.5 s to pack layers at the 99th percentile (879.9 MB) while it takes, on average, 1,853.9 s to unpack the layers.

Packing/unpacking time also increases with layer size. For example, when an SSD is used to store the unpacked layer content the unpacking time increases from 0.1 s to 520.2 s as the layer size increases from 3 KB to 878.9 MB. Furthermore, using memory to save the unpacked layer content can significantly reduce packing/unpacking time. Packing time decreases by up to 8.6× and 17.9× compared to using a SSD or a HDD, respectively, while for unpacking, the time reduction is up to 129.5× and 461.5×. Therefore, using memory, such as a RAM disk, to temporally host the unpacked layer content and lazily write them back to persistent storage can efficiently reduce packing/unpacking time during layer pushing/pulling. Moreover, the tar archiving process sequentially stores the files in the output tarball. Thus, if a layer is significantly large, parallelizing the archiving process can greatly reduce the packing/unpacking overhead.

4.3.2 Storage option impact on packing and unpacking

Figure 4.3: Storage option impact.

Figure 4.4: Concurrency impact. X-axis shows the layer size percentile and number of concurrent threads.

(a) Compression/decompression and network latency.

(b) Packing/unpacking latency.
4.3.3 Concurrency impact

Figure 4.4(a) shows the impact of concurrent pulling/pushing threads on compression, decompression, and network transfer. When the concurrency level increases, layer compression time, decompression time, and network transfer time increase slightly. For example, when the number of concurrent layer pulling/pushing threads increases from 2 to 4, the decompression time increases by 1.1–1.3× across different layer sizes and the compression time increases by 1.1–1.4×. The network transfer time also increases by 1.0–1.4×. Moreover, we observe that decompression and compression take more time than network transfer under concurrent layer pulling/pushing threads due to IO contention.

Figure 4.4(b) shows the concurrency impact on packing and unpacking when a SSD is used to store uncompressed and unpacked layer contents. When the concurrency level increases from 2 to 4, the time to pack and unpack a layer increase by 1.3–2× and 1.3–1.6× respectively. Consequently, using SSDs or HDDs to store unpacked layer contents under concurrent layer pulling/pushing will incur a considerable overhead on packing/unpacking. However, using a RAM disk to host unpacked layer contents under concurrent layer pulling/pushing will surely consume a large amount of memory space. Therefore, using a small memory as a packing and unpacking cache and gradually writing the unpacked layer contents to SSDs/HDDs can significantly reduce packing/unpacking time.

4.4 Container Storage Driver Performance Analysis

As a major component of containers, the I/O performance of container storage drivers is critical to the image build time, container startup time, and container execution time. Therefore, in this section, we evaluate the I/O performance of multiple widely used container storage drivers. We mainly focus on small block sizes for a couple of reasons. First, during image building, commands like COPY, RUN apt install, or RUN git clone can write files into layers inside building containers. Since there are many small-sized files stored in layers and therefore images (i.e. 50% of the files are smaller than 4 KB as detailed in §?? and Table 4.1), the I/O performance for small I/O requests (i.e. small block sizes) is important for the image building latency.

Second, commands like RUN make builds (i.e. compiles, archives or links), and writes executables from source code into layers inside building containers as containerized executables. The block sizes involved in executable building usually range from few bytes to 1 MB, e.g., half of I/O requests’ block sizes are smaller than 4 KB [118].

We aim to address the following research questions.

1. What is the I/O performance distribution across various block sizes? How does block size impact the read/write performance of containers?
Table 4.2: Schemes.

<table>
<thead>
<tr>
<th>Raw file system</th>
<th>Container storage drivers</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Drivers</td>
</tr>
<tr>
<td>Xfs</td>
<td>Overlay2</td>
</tr>
<tr>
<td>Btrfs</td>
<td>Btrfs</td>
</tr>
<tr>
<td>Device Mapper + Xfs</td>
<td>Devicemapper</td>
</tr>
</tbody>
</table>

2. What is the overhead caused by container storage drivers? How does the performance of storage drivers compare?

3. What is the impact of concurrency on the I/O performance of container storage drivers?

**Testbed.** Our testbed is detailed in Section 4.3. We use SSDs as storage drives for testing container drivers. To get a stable SSD read/write performance, we first use the dd command to write a large amount of data to SSDs with a block size of 1 GB sequentially until the write performance stabilizes at 230 MB/s. Moreover, we use flexible I/O tester fio to measure the performance of Docker container storage drivers. Before each test, we clear PageCache, delete the data written by the previous test, set O_DIRECT flag and use asynchronous I/O engine libaio for fio, and disable write buffers on storage drives.

**Schemes.** We study three popular container storage drivers as follows: Overlay2, Btrfs, and Devicemapper. The backend file systems for the above drivers are Xfs, Btrfs, and Device Mapper (DM) thin pools in direct-lvm mode with Xfs, respectively as shown in Table 4.2.

To evaluate the overhead caused by container storage drivers, we compare each storage driver with its corresponding backend file system without using a container, denoted as raw file system, as shown in Table 4.2. Each backend file system and raw file system are created on a separate physical partition on SSD drives.

**4.4.1 Small I/O requests**

To evaluate the I/O performance of different storage drivers for varying block sizes, we launch a fio container by using a fio image. We first create 1000 128 KB files and measure the random I/O performance using different block sizes.

Figure 4.5 shows IOPS, bandwidth, and average I/O completion latency for random reads. We observe that the raw file systems have a higher IOPS and bandwidth and lower I/O completion latency compared to their corresponding container storage drivers. For example, the latencies of Overlay2, DM, and Btrfs are 1.01×, 1.2×, and 3.7× higher than their corresponding raw file systems: Xfs-RAW, DM-RAW, and Btrfs-RAW, respectively, when the block size is 2 KB.
Figure 4.5: I/O size impact on reads. X-axes represents I/O size.

Figure 4.6: I/O size impact on writes. X-axes represents I/O size.
Moreover, when block sizes are smaller than 4 KB, Btrfs exhibits the highest IOPS, bandwidth, and lowest I/O completion latency for both the storage driver and the raw file system. For example, when the block size is 2 KB, the IOPS for Btrfs and Btrfs-RAW are 23.1 K and 25.4 K, respectively, as shown in Figure 4.5(a). The bandwidth of Btrfs is 1.5× and 1.9× higher than Overlay2 and DM, respectively, as shown in Figure 4.5(b). The latency of Btrfs is 20.8× and 25× lower than the latency Overlay2 and DM, respectively, as shown in Figure 4.5(c). And the bandwidth of Btrfs-RAW is 1.5× and 1.6× higher than Xfs-RAW and DM-RAW. Similarly for Btrfs-RAW, its latency is 76.6× and 79.2× lower than Xfs-RAW and DM-RAW as shown in Figure 4.5(c).

However, when the block size increases, both Overlay2 and Xfs-RAW show slightly higher IOPS and bandwidth and lower I/O completion latency than the other four schemes as shown in Figure 4.5. Overall, Figure 4.5 indicates that during image building or container execution, if the I/O requests’ block sizes are smaller than 4 KB, Btrfs has the best small file read performance compared to Overlay2 and DM. Otherwise, Overlay2 can provide a better read performance for containerized applications.

As for write performance, Btrfs performs slightly better than Overlay2 and DM when the block size is smaller than 4 KB as shown in Figure 4.6. For example, the bandwidth of Btrfs is 1.1× and 1.3× higher than the bandwidth of Overlay2 and DM, respectively. While when the block size increases, Overlay2 outperforms both Btrfs and DM.

4.4.2 CoW performance

To evaluate the overhead caused by the CoW mechanism of container storage drivers, we launch a fio container and create a new layer by randomly creating and writing 1000 128 KB files and measure random write performance, denoted as creation performance. After that, we commit the container as a new image and run the newly created image as a container instance. We then randomly rewrite 6 KB data to each file in the preceding read-only layer, and measure the rewrite performance to the preceding read-only layer files, denoted as rewrite performance.

We observe that when the block size is bigger (i.e. block size ≥ 4 KB), container storage drivers and their corresponding raw file systems exhibit similar performance. Therefore, in this experiment, we set a smaller block size for I/O requests to 2 KB.

Figure 4.7 shows the IOPS and bandwidth for both layer file creation write requests and rewrite requests to the read-only layer files. Overall, the creation write performance is slightly higher than the rewrite performance for all the container drivers. For example, IOPS and bandwidth degrade 4%-23%, for rewrites compared to creation writes. Btrfs degrades the most for rewrites compared to Overlay2 and DM, though the performance of Btrfs is best in both creation and rewriting. For raw file systems the rewrite performance is slightly higher than creation performance, with the exception of Btrfs-RAW as shown in
Moreover, we observe that raw file systems outperform container storage drivers. For example, with Btrfs container storage driver, the IOPS degrade by 11.3% compared to Btrfs-RAW for creation. It indicates that container storage drivers incur rewrite overhead when the block size is smaller.

4.4.3 Concurrency impact

To study the concurrency impact on the performance of container storage drivers, we increase the number of read/write threads for fio and show their I/O latency in Figure 4.8. Note that the block size is set to 2 KB same as §4.4.2.

Read latency increases with concurrency as shown in Figure 4.8. The I/O latency increases 1.02-1.5× as the number of read threads increases from 2 to 5. We observe that Xfs-RAW and Overlay2 are more sensitive to concurrency. They increase by up to 1.5× and 1.14× as concurrency increases from 2 to 5, respectively. The other four schemes only increase by 1.02-1.06× and remain almost stable while the number of read threads increases.
As shown in Figure 4.8, the write completion latency is more stable with concurrency than read latency. The I/O latency increases $1.04-1.2 \times$ as the number of write threads increases from 2 to 5. For example, the I/O latency of DM-RAW increases up to $1.2 \times$ while for the remaining schemes, it increases by less than $1.09 \times$.

### 4.4.4 Analyses Summary

We summarize the layer pulling latency (§4.3) and container storage drivers (§4.4) analyses in five observations:

1. Compression/decompression can become a bottleneck for layer pulling/pushing. Parallel compression algorithms such as Pigz and Lz4 can significantly reduce compression/decompression time.

2. Packing/unpacking can become a bottleneck for layer pulling/pushing. Using memory as a cache to temporally store unpacked layer content can significantly reduce the latency.

3. To handle I/O requests, Btrfs performs better than both DM and Overlay2 for small reads (i.e. block size < 4 KB), whereas Overlay2 gives the best performance with big reads (i.e. block size $\geq 4$ KB).

4. When block sizes are small, the performance of container storage drivers is lower than their corresponding raw backend file system (without using containers). This means that container storage drivers add an additional overhead on their corresponding backend file systems.

5. Rewrites to files in preceding read-only layers is slower than writing new files, for all container storage drivers evaluated (i.e. Overlay2, Btrfs, and DM).

### 4.5 Registry with Deduplication Support

In this section, we first describe a high-level design of — a Docker registry that supports file-level deduplication. We then proceed with a simulation-based evaluation of the expected performance implications.

#### 4.5.1 Design

We designed so that the interface between the Docker clients and the registry remains unchanged. As such, no modifications to the Docker clients are needed. Below we describe how handles layer pushes and pulls at the registry side.
Chapter 4. Design Implications for Container Storage Systems via Performance Analysis

Push handles push requests asynchronously. After receiving a layer from a client, it does not immediately unpack the layer. Instead, it reliably stores the layer’s compressed tarball in a persistent staging area. A separate off-line deduplication process iterates over the layers in the staging area and performs the following steps for every layer:

1. decompress and unpack the layer’s tarball into individual files;
2. compute a fingerprint for every file in the layer;
3. check all file fingerprints against the file index to identify if identical files are already stored in;
4. store non-deduplicated files in ’s storage;
5. create and store a layer recipe that includes the path, metadata, and fingerprint of every file in the layer;
6. remove the layer’s tarball from the staging area.

Layer recipes are identified by layer digests and files are identified by their fingerprints. These identifiers are used to address corresponding objects in the underlying storage. For example, if a file system is used as a backend storage, creates a single file for every layer recipe (named by the digest) and a single file for every in-layer file (named by the fingerprint).

Pull When a layer is pulled, has to reconstruct the layer based on the layer recipe. A pull request cannot be postponed to an off-line process as the pulling client is actively waiting for the layer. performs the following steps inline during the pull request:

1. check if the requested layer is still in the staging area and if so, service it directly from there;
2. otherwise, find the layer recipe by the layer digest provided by the client;
3. prepare a directory structure for the layer based on the layer recipe;
4. pack and compress the layer’s directory tree into a temporary tarball;
5. send the layer tarball back to the client and then discard the layer tarball.

4.5.2 Performance evaluation

While can effectively eliminate redundant files in the Docker registry, it introduces overhead which can reduce the registry’s performance.
Simulation To analyze the impact of file-level deduplication on performance, we conduct a preliminary simulation-based study of. Our simulation approximates several of’s steps. First, a layer from our dataset is copied to a RAM disk. The layer is then decompressed, unpacked, and the fingerprints of all files are computed using the MD5 hash function [161]. The simulation searches the fingerprint index for duplicates, and, if the file has not been stored previously, it records the file’s fingerprint in the index. At this point our simulation does not include the latency of storing unique files. To simulate the layer reconstruction during a pull request, we archive and compress the corresponding files.

The simulator is implemented in 600 lines of Python code and our setup is a one-node Docker registry on a machine with 32 cores and 64 GB of RAM. To speed up the experiments and fit the required data in RAM we use 50% of all layers and exclude the ones larger than 50 MB. We process 60 layers in parallel using 60 threads. The entire simulation took 3.5 days to finish.

Figure 4.9 shows the CDF for each sub-operation of. Unpacking, Decompression, Digest Calculation, and Searching are part of the deduplication process and together make up the Dedup time. Searching, Archiving, and Compression simulate the processing for a pull request and form the Pulling time.

Push does not directly impact the latency of push requests because deduplication is performed asynchronously. The appropriate performance metric for push is the time it takes to deduplicate a single layer. Looking at the breakdown of the deduplication time in Figure 4.9, we make several observations.

First, the searching time is the smallest among all operations with 90% of the searches completing in less than 4 ms and a median of 3.9 ms. Second, the calculation of digests spans a wide range from 5 µs to almost 125 s. 90% of digest calculation times are less than 27 s while 50% are less than 0.05 s. The diversity in the timing is caused by a high variety of layer sizes both in terms of storage space and file counts. Third, the run time for decompression and unpacking follows an identical distribution for around 60% of the layers and is less than 150 ms. However, after that, the times diverge and decompression times increase faster compared to unpacking times. 90% of decompressions take less than 950 ms while 90% of
packing time is less than 350ms.

Overall, we see that 90% of file-level deduplication time is less than 35 s per layer, while the average processing time for a single layer is 13.5 s. This means that our single-node deployment can process about 4.4 layers/s on average (using 60 threads). In the future we will work on further improving ’s deduplication throughput.

Pull From Figure 4.9 we can see that 55% of the layers have close compression and archiving times ranging from from 40 ms to 150 ms and both operations contribute equally to pulling latency. After that, the times diverge and compression times increase faster with an 90th percentile of 8 s. This is because compression times increase for larger layers and follow the distribution of layer sizes (see Figure 3.3). Compression time makes up the major portion of the pull latency and is a bottleneck. Overall, the average pull time is 2.3 s.

4.5.3 Enhancements

We propose additional optimizations that can help to speed up:

1. As the majority of the pull time is caused by compression, we propose to cache hot layers as precompressed tar files in the staging area. According to our statistics, only 10% of all images were pulled from Docker Hub more than 360 times from the time the image was first pushed to Docker Hub until May 30, 2017. Moreover, we found that 90% of pulls went to only 0.25% of images based on image pull counts. This suggests the existence of both cold and hot images and layers.

2. As deduplication provides significant storage savings, can use faster but less effective local compression methods than gzip [18].

3. The registries often experience fluctuation in load with peaks and troughs [48]. Thus, file-level deduplication can be triggered when the load is low to prevent interference with client pull and push requests.

4.6 Chapter Summary

In this chapter, we investigated the I/O performance of storage drivers based on our image characterizations on a compressed 47 TB dataset with over 1 million layers and 5 billion files. Starting with layer pulling/pushing analysis, we found that compression methods and storage options for unpacked layer contents can both become bottlenecks. Various optimizations are proposed to speed up layer pulling/pushing. Moreover, we drew several design implications by analyzing container storage drivers using a wide range of block sizes.
We proceeded with a simulation-based evaluation of the impact of deduplication on the Docker registry performance. We found that restoring large layers from registry can slow down pull performance due to compression overhead. To speed up, we suggested several optimizations. Our findings justify and lay way for integrating deduplication in the Docker registry.
Chapter 5

Flexible High-Performance Deduplication for Docker Registries

5.1 Introduction

Containerization frameworks such as Docker [6] have seen a remarkable adoption in modern cloud environments. This is due to their lower overhead compared to virtual machines [28, 123], their rich ecosystem to ease application development, deployment, and management [80], and the growing popularity of microservices [187]. By now, all major cloud platforms endorse containers as a core deployment technology [31, 97, 104, 135] and container adoption is increasing. For example, Datadog reports that in 2018, about 21% of its customers’ monitored hosts ran Docker and that this trend continues to grow by about 5% annually [83].

Container images are at the core of containerized applications. An application’s container image includes the executable of the application along with a complete set of its dependencies—other executables, libraries, and configuration and data files. Images are structured in layers. When building an image with Docker, each executed command, such as `apt install`, creates a new layer on top of the previous one [9], which contains the files that the command has modified or added. Docker leverages union file systems [177] to efficiently merge layers into a single file system tree when starting a container. Containers can share identical layers across different images.

To store and distribute container images, Docker relies on image registries (e.g., Docker Hub [7]). Docker clients can push images to or pull them from the registries as needed. On the registry side, each layer is stored as a compressed tarball and identified by a content-based address. The Docker registry supports various storage backends for saving and retrieving layers. For example, a typical large-scale setup stores each layer as an object in an object store [105, 142].
As the container market continues to expand, Docker registries have to manage a growing number of images and layers. Some conservative estimates show that in spring 2019, Docker Hub alone stored at least 2 million public images totaling roughly 1 PB in size [167, 196]. We believe that this is just the tip of the iceberg and the number of private images is significantly higher. Other popular public registries [30, 49, 50, 96, 108, 134], as well as on-prem registry deployments in large organizations, experience a similar surge in the number of images. As a result, organizations spend an increasing amount of their storage and networking infrastructure on operating image registries.

The storage demand for images is worsened by the large amount of duplicate data in images. As Docker images must be self-contained by definition, different images frequently include the same, common dependencies (e.g., libraries). As a result, different images are prone to contain a high number of duplicate files as shared components exist in more than one image.

To reduce this redundancy, Docker employs layer sharing. However, this is insufficient as layers are coarse and rarely identical because they are built by developers independently and without coordination. Indeed, a recent analysis of the Docker Hub image dataset showed that about 97% of files across layers are duplicates [196]. Registry storage backends exacerbate the redundancy further due to the replication they perform to improve image durability and availability [57].

Deduplication is an effective method to reduce capacity demands of intrinsically redundant datasets [156]. However, applying deduplication to a Docker registry is challenging due to two main reasons: 1) layers are stored in the registry as compressed tarballs, which do not deduplicate well [132]; and 2) decompressing layers first and storing individual files incurs high reconstruction overhead and slows down image pulls. The slowdowns during image pulls are especially harmful because they contribute directly to the start times of containers. Our experiments show that, on average, naive deduplication increases layer pull latencies by up to 98× compared to a registry without deduplication.

In this chapter, we propose Duphunter, the first Docker registry that natively supports deduplication. Duphunter is designed to increase storage efficiency via layer deduplication while reducing the corresponding layer restoring overhead. It utilizes domain-specific knowledge about the stored data and the storage system to reduce the impact of layer deduplication on performance. For this purpose, it combines five key techniques:

1. Duphunter exploits existing replication to improve performance. It keeps a specified number of layer replicas as-is, without decompressing and deduplicating them. Accesses to these replicas do not experience layer restoring overhead. Any additional layer replicas needed to guarantee the desired availability are decompressed and deduplicated.

2. Duphunter deduplicates rarely accessed layers more aggressively than popular ones to speed up accesses to popular layers and achieve higher storage savings.
3. By monitoring user access patterns, Duphunter proactively restores layers before layer download requests arrive, avoiding reconstruction latency during pulls.

4. To parallelize and speed up layer reconstruction, Duphunter groups files from a single layer in slices and evenly distributes them across the cluster.

5. By exploiting the flexibility of Duphunter, we provide the first comprehensive analysis of the impact of different deduplication levels (file and block) and redundancy policies (replication and erasure coding) on registry performance and space savings.

We evaluate Duphunter on a 6-node cluster using real-world workloads and layers. In the highest performance mode, Duphunter outperforms the state-of-the-art by reducing layer pull latencies by up to 2.8×. In the highest deduplication mode, Duphunter reduces storage consumption by up to 6.9×. The other deduplication modes make various tradeoffs in performance and space savings.

5.2 Background

We first provide the background on the Docker registry and then discuss existing deduplication work.

5.2.1 Docker Registry

The main purpose of a Docker registry is to store and distribute container images to Docker clients. A registry provides a REST API that allows Docker clients to push images to and pull images from it [87, 88]. Docker registries group images into repositories, each containing versions (tags) of the same image, identified as <repo-name:tag>. For each tagged image in a repository, the Docker registry stores a manifest, i.e. a JSON file, which contains the runtime configuration for a container image (e.g., environment variables) and the list of layers that make up the image. A layer is stored as a compressed archival file and identified using a digest (SHA-256), computed over the uncompressed content of the layer. When pulling an image, a Docker client first downloads the manifest and then the referenced layers, which are not already present on the client. When pushing an image, a Docker client first uploads the layers (if not already present in the registry) and then the manifest.

The current Docker registry software is a single-node application with a RESTful API. The registry delegates storage to a backend storage system, ranging from local file systems to distributed object storage systems such as Swift [142] or others [2, 3, 105, 142] through the corresponding storage drivers. To scale the registry, organizations typically deploy a load balancer or proxy in front of several independent registry instances [48]. In this case, client requests are forwarded to the destination registries through a proxy, then served by
5.2. Background

the registries’ backend storage system. To reduce the communication overhead between the proxy, registry, and backend storage system, Bolt[127] proposes to use a consistent hashing function instead of a proxy, distribute requests to registries, and utilize the local file system on each registry node to store data instead of using a remote distributed object storage system. Multiple layer replicas are stored on Bolt registries for high availability and reliability. Duphunter is implemented based on the architecture of Bolt registry for high scalability.

Registry performance is critical to Docker clients. In particular, the layer pulling performance (i.e. GET layer performance) impacts container startup times [102] significantly. Related work has studied various dimensions of registry performance and a Docker image dataset [48, 64, 102, 170, 177, 192, 196]. However, existing work does not provide deduplication capabilities to the registry. A community proposal exists to add file-level deduplication to container images [29], but as of now lacks a detailed design or performance analysis. Skourtis et al. [167] propose restructuring layers to optimize for various dimensions, including registry storage utilization. Their approach does not remove all duplicates, whereas Duphunter leaves images unchanged and can eliminate all duplicates in the registry. Much work aims to reduce the size of a single image [89, 99, 159, 179], which is complementary to Duphunter.

5.2.2 Deduplication

Data deduplication has received considerable attention, particularly for virtual machine images [106, 111, 171, 199]. Many deduplication studies focus on primary and backup data deduplication [91, 92, 93, 125, 126, 128, 138, 163, 178, 184, 200] and show the effectiveness of file- and block-level deduplication [131, 173]. To further reduce storage space, block-level deduplication integrating with compression is proposed [181]. In addition to local deduplication schemes, a global deduplication method [140] is proposed to improve the deduplication ratio and provide high scalability for distributed storage systems.

Data restoring latency is an important factor for storage systems with deduplication support. Efficient chunk caching algorithms and forward assembly are proposed to accelerate data restore performance [66]. At first glance, one could apply existing deduplication techniques to solve the issue of high data redundancy among container images. However, as we demonstrate in detail in §5.3.2, such a naive approach leads to slow reconstruction of layers on image pulls, which severely degrades container startup times. Duphunter is specifically designed for Docker registries, which allows it to leverage image and workload information to reduce deduplication and layer restore overhead.
5.3 Motivating Observations

The need and feasibility of Duphunter is based on three key observations: 1) container images experience a lot of redundancy; 2) existing scalable deduplication technologies significantly increase image pull latencies; 3) image access patterns can be predicted reliably.

5.3.1 Redundancy in Container Images

Container image layers exhibit a large degree of redundancy in terms of duplicate files. Although Docker supports the sharing of layers among different images to remove some redundant data in the Docker registry, this is not sufficient to effectively eliminate duplicates. According to the deduplication analysis of the Docker Hub dataset [196], 97% of files have more than one file duplicate, resulting in a deduplication ratio of $2 \times$ in terms of capacity. We believe that the deduplication ratio is much higher when private repositories are taken into account.

The duplicate files are executables, object code, libraries, and source code, and are likely imported by different image developers using package installers or version control systems such as apt, pip or git to install similar dependencies. However, as layers often share many but not all files, this redundancy cannot be eliminated by Docker’s current layer sharing approach.

$R$-way replication for reliability additionally fuels the high storage demands of Docker registries. Hence, satisfying demand by adding more disks and scaling out storage systems quickly becomes expensive.
5.3. Motivating Observations

Table 5.1: Dedup. ratio vs GET layer latency increases

<table>
<thead>
<tr>
<th>Technology</th>
<th>Dedup ratio, compressed layers</th>
<th>Dedup ratio, uncompressed layers</th>
<th>GET latency increase, uncompressed layers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jdupes</td>
<td>1</td>
<td>2.1</td>
<td>36 ×</td>
</tr>
<tr>
<td>VDO</td>
<td>1</td>
<td>4</td>
<td>60 ×</td>
</tr>
<tr>
<td>Btrfs</td>
<td>1</td>
<td>2.3</td>
<td>51 ×</td>
</tr>
<tr>
<td>ZFS</td>
<td>1</td>
<td>2.3</td>
<td>50 ×</td>
</tr>
<tr>
<td>Ceph</td>
<td>1</td>
<td>3.1</td>
<td>98 ×</td>
</tr>
</tbody>
</table>

5.3.2 Drawbacks of Existing Technologies

A naive approach to eliminating duplicates in container images could be to apply an existing deduplication technique. To experimentally demonstrate that such a strategy has significant shortcomings, we try four popular local deduplication technologies, VDO [182], Btrfs [61], ZFS [191], Jdupes [107], in a single-node setup and one distributed solution, Ceph [68], on a 3-node cluster. The deduplication block sizes are set to 4 KB for both VDO and Ceph, and 128KB for both Btrfs [61] and ZFS [191] as default. Note that compression is also enabled for VDO as default. Table 5.1 summarizes the deduplication ratios and pull latency overhead for each technology. (The details of the dataset and testbed are presented in §5.5.)

Deduplication ratios Putting the original compressed layer tarballs in any of the deduplication systems results in a deduplication ratio of 1 because compressed data hardly duplicates [132]. Hence, to expose the redundancy to the deduplication systems, we decompress every layer before storing it.

After decompression, all deduplication schemes yield significant deduplication ratios. Jdupes, Btrfs, and ZFS reduce the dataset to about half and achieve deduplication ratios of 2.1, 2.3, and 2.3, respectively. Ceph has a higher deduplication ratio since it uses a smaller deduplication block size while VDO shows the highest deduplication ratio as it also compresses deduplicated data.

It is important to note that for an enterprise-scale registry a large number of storage servers need to be deployed and single-node deduplication systems (Jdupes, Btrfs, ZFS, and VDO) can only deduplicate data within a single node. Therefore, in a multi-node setup, they can never achieve optimal global deduplication, i.e. duplicates will occur across nodes.

Pull latencies To analyze layer pull latencies, we implement a layer restoring process for each technology. Restoring includes fetching files, creating a layer tarball, and compressing it. We measure the average GET layer latency and calculate the restore overhead compared
to GET requests without layer deduplication.

As shown in Table 5.1, the restoration overhead is high. The file-level deduplication scheme Jdupes increases the GET layer latency by $36 \times$. This is caused by the expensive restoring process. Btrfs, ZFS, and VDO show an increase of more than $50 \times$ as they are block-level deduplication systems and hence, they also add file restoring overhead. The overhead for Ceph is the highest because restoration is distributed and incurs network communication.

In summary, our analysis shows that while existing technologies can provide storage space savings for container images (after decompression), they incur high cost during image pulls due to slow layer reconstruction. At the same time, pull latency constitutes the major portion of container startup times even without deduplication [102] and hence, should not be increased.

5.3.3 Predictable User Access Patterns

A promising approach to mitigate layer restoring overhead is predicting which layers will be accessed and preconstruct them. In Duphunter, we can exploit the fact that when a Docker client pulls an image from the registry, it first retrieves the image manifest, which includes references to the image layers.

User pulling patterns  Typically, if a layer is already stored locally, then the client will not fetch this layer again. However, higher-level container orchestrators allow users to configure different policies for starting new containers. For example, Kubernetes allows policies such as IfNotPresent, i.e. only get the layer if it hasn’t been pulled already, or AlwaysGet, i.e. always retrieve the layer, even if it is already present locally. These different behaviors need to be considered when predicting whether a layer will be pulled by a user or not.

We use the IBM Cloud registry workload [48] to analyze the likelihood for a user to repull an already present layer. The traces span ~80 days for 7 registry clusters: Dallas, Frankfurt, London, Sydney, Development, Prestaging, and Staging. Figure 5.1 shows the CDF of layer GET counts by the same clients. The analysis shows that the majority of layers are only fetched once by the same clients. For example, 97% of layers from Syd are only fetched once by the same clients. However, there are clients that pull the same layers repeatedly. E.g., a client from London fetched the same layer 19,300 times.

Figure 5.2 shows the corresponding client repull probability, calculated as the number of repulled layers divided by the number of total GET layer requests issued by the same client. We see that 50% of the clients have a repull probability of less than 0.2 across all registries. We also observe that the slope of the CDFs is steep at both lower and higher probabilities but becomes flat in the middle. This suggests that, by observing access patterns, we are able to classify clients into 2 classes, always-pull clients and pull-once clients, and predict, whether they will pull a layer or not by keeping track of user access history.
5.4. Duphunter Design

Layer preconstruction  We analyze the inter-arrival time between a GET manifest request and the subsequent GET layer request. As shown in Figure 5.3, the majority of intervals are greater than 1 second. For example, 80% of intervals from London are greater than 1 second and 60% of the intervals from Sydney are greater than 5 seconds.

There are several reasons for this long gap. First, when fetching an image from a registry, the Docker client fetches a fixed number of layers in parallel (three by default) starting from the lowest layer. In the case where an image contains more than three layers, the upper layers have to wait until the lower layers are downloaded, which delays the GET layer request for these layers. Second, network delay between clients and registry often accounts for a large portion of the GET latency in cloud environments.

As we show in §5.5, layer preconstruction can significantly reduce layer restoring overhead. In the case of a shorter duration between a GET manifest request and its subsequent GET layer requests, layer preconstruction can still be beneficial because the layer construction starts prior to the arrival of the GET request.

5.4  Duphunter Design

In this section, we first provide an overview of Duphunter (§5.4.1). We then describe in detail how it deduplicates (§5.4.2) and restores (§5.4.3) layers, and how it further improves performance via predictive cache management (§5.4.4). Finally, we discuss the integration of sub-file deduplication and erasure coding with Duphunter (§??).
5.4.1 Overview

Figure 5.4 shows the architecture of Duphunter. Duphunter consists of two main components: 1) a cluster of storage servers, each exposing the registry REST API and 2) a distributed metadata database. When uploading or downloading layers, Docker clients communicate with any Duphunter server using the registry API. Each server in the cluster contains an API service and a backend storage system. The backend storage systems store layers and perform deduplication, keeping the deduplication metadata in the database. Duphunter uses three techniques to reduce deduplication and restoring overhead: 1) replica deduplication modes; 2) parallel layer reconstruction; and 3) proactive layer prefetching/preconstruction.

**Replica deduplication modes** For higher fault tolerance and availability, existing registry setups replicate layer files. Duphunter also performs layer replication, but additionally allows to deduplicate files inside the replicas.

A basic deduplication mode \( n \) (B-mode \( n \)) defines that Duphunter should only keep \( n \) layer replicas intact and deduplicate the remaining \( R - n \) layer replicas, where \( R \) is the layer replication level. At one extreme, B-mode \( R \) means that no replicas should be deduplicated and hence, provides the best performance but no data reduction. At the other end, B-mode 0 deduplicates all layer replicas, i.e. it provides the highest deduplication ratio but adds restoration overhead for GET requests. The remaining B-modes in between allow to trade off performance for data reduction.

For heavily skewed workloads, Duphunter also provides a selective deduplication mode (S-mode). The S-mode utilizes the skewness in layer popularity, observed in [48], to decide how many replicas should be deduplicated for each layer. As there are hot layers that are pulled frequently, S-mode sets the number of intact replicas proportional to their popularity. This means that hot layers have more intact replicas and hence, can be served faster, while cold layers are deduplicated more aggressively.

Deduplication in Duphunter, for the example of B-mode 1, works as follows: Duphunter first creates 3 layer replicas across 3 servers. It keeps a single layer replica as the primary layer replica on one server. Deduplication is then carried out in one of the other servers storing a replica, i.e. the layer replica is decompressed and any duplicate files are discarded while unique files are kept. The unique files are replicated and saved on different servers. Once deduplication is complete, the remaining two layer replicas are removed. Any subsequent GET layer requests are sent to the primary replica server first since it stores the complete layer replica. If that server crashes, one of the other servers is used to rebuild the layer and serve the GET request.

To support the different deduplication modes, Duphunter divides storage servers into two groups (Figure 5.4): a primary cluster consisting of P-servers and a deduplication cluster consisting of D-servers. P-servers are responsible for storing full layer replicas and replicas of
5.4. Duphunter Design

the manifest, while D-servers deduplicate layer replicas at the file level, store the unique files, and replicate them. The split ensures that layer replicas and their containing file replicas are stored on different servers to maintain fault tolerance.

P- and D-servers form a 2-tier storage hierarchy. In the default case, the primary cluster serves all incoming GET requests. If a request cannot be served from the primary cluster (e.g., due to a node failure, or Duphunter operating in B-mode 0 or S-mode), it will be forwarded to the deduplication cluster and the requested layer will be reconstructed.

**Parallel layer reconstruction**  Duphunter speeds up layer reconstruction through parallelism. As shown in Figure 5.4, each D-server’s local storage is divided into three parts: the layer stage area, preconstruction cache, and file store. The layer stage area temporarily stores newly added layer replicas. After deduplicating a replica, the resulting unique files are stored in a content addressable file store and replicated to the peer servers to provide redundancy. Once all file replicas have been stored, the layer replica is deleted from the layer stage area.

Duphunter distributes the layer’s unique files onto several servers (see §5.4.2). All files on a single server belonging to the same layer are called a slice. A slice has a corresponding slice recipe, which defines the files that are part of this slice, and a layer recipe defines the slices needed to reconstruct the layer. This information is stored in Duphunter’s metadata database. This allows D-servers to rebuild layer slices in parallel and thereby improve reconstruction performance. Duphunter maintains layer and file fingerprint indices in the metadata database.

**Predictive cache prefetch and preconstruction**  To improve the layer access latency, Duphunter employs a cache layer in both the primary and the deduplication clusters, respectively. Each P-server has an in-memory user-behavior based prefetch cache to reduce disk I/Os. When a GET manifest request is received from a user, Duphunter predicts which layers in the image will actually need to be pulled and prefetches them in the cache. Additionally, to reduce layer restoring overhead, each D-server maintains an on-disk user-behavior based preconstruct cache. As with the prefetch cache, when a GET manifest request is received, Duphunter predicts which layers in the image will be pulled, preconstructs the layers, and loads them in the preconstruct cache. To accurately predict which layers to prefetch, Duphunter maintains two maps: ILmap and ULmap. ILmap stores the mapping between images and layers while ULmap keeps track of a user’s access history, i.e. which layers the user has pulled and how many times (see §5.4.4).
5.4.2 Deduplicating Layers

As in the traditional Docker registry, Duphunter maintains a *layer index*. After receiving a PUT layer request, Duphunter first checks the layer fingerprint in the *layer index* to ensure an identical layer is not already stored. If not, Duphunter, replicates the layer \( r \) times across the P-servers and submits the remaining \( R - r \) layer replicas to the D-servers. Those replicas are temporarily stored in the layer stage areas of the D-servers. Once the replicas have been stored successfully, Duphunter notifies the client of the request completion.

**File-level deduplication** Once in the staging area, one of the D-servers decompresses the layer and starts the deduplication process. First, it extracts file entries from the tar archive. Each file entry is represented as a *file header* and the associated *file content* [95]. The file header contains metadata such as file name, path, size, mode, owner information, etc. Duphunter records every file header in slice recipes (described below) to be able to correctly restore the complete layer archive later.

To deduplicate a file, Duphunter computes a file Id by hashing the file content and checks if the Id is already present in the file index. If present, the file content is discarded. Otherwise, the file content is assigned to a D-server and stored in its file store, and the file Id is recorded in the file index. The file index maps different file Ids to their physical replicas stored on different D-servers.

**Layer partitioning** Duphunter picks D-servers for files to improve reconstruction times. For that, it is important that different layer slices are similarly sized and evenly distributed across D-servers. To achieve this, Duphunter employs a greedy packing algorithm. Consider first the simpler case in which each file only has a single replica. Duphunter first computes the total size of the layer’s existing shared files on each D-server (this might be 0 if a D-server does not store any shared files for the layer). Next, it assigns the largest new unique file to the smallest partition until all the unique files are assigned. Note that during layer partitioning, Duphunter does not migrate existing shared files to reduce I/O overhead.

In the case where a file has more than one replica, Duphunter performs the above-described partitioning *per replica*. That means that it first assigns the primary replicas of the new unique files to D-servers according to the location of the primary replicas of the existing shared files. It then does the same for the secondary replicas and so on. Duphunter also ensures that two replicas of the same file are never placed on the same node.

**Unique file replication** Next, Duphunter replicates and distributes the unique file replicas across D-servers based on the layer partitioning. The headers and content pointers of all files in the deduplicated layer that are assigned to a specific D-server are included in that D-server’s *slice recipe* for that layer. After file replication, Duphunter adds the new slice
5.4. Duphunter Design

Duphunter also creates a layer recipe for the uploaded layer and stores it in the metadata database. The layer recipe records all the D-servers that store slices for that layer and which can act as restoring workers. When a layer needs to be reconstructed, one worker is selected as the restoring master, responsible for gathering all slices and rebuilding the layer (see §5.4.3).

Figure 5.5 shows an example deduplication process. The example assumes B-mode 1 with 3-way replication, i.e. each unique file has two replicas distributed on two different D-servers. The files $f_1$, $f_2$, and $f_3$ are already stored in Duphunter, and $f_1'$, $f_2'$, and $f_3'$ are their corresponding replicas. Layer $L_1$ is being pushed and contains files $f_1$–$f_6$. $f_1$, $f_2$, and $f_3$ are shared files between $L_1$ and other layers, and hence, are discarded during file-level deduplication. The unique files $f_4$, $f_5$ and $f_6$ are added to the system and replicated to D-servers $A$, $B$, and $C$.

After replication, server $B$ contains $f_2$, $f_5$, $f_1'$, and $f_4'$. Together $f_2$ and $f_5$ form the primary slice of $L_1$, denoted as $L_1 :: B :: P$. This slice Id contains the layer Id the slices belongs to ($L_1$), the node, which stores the slice ($B$) and the backup level ($P$ for primary). The two backup file replicas $f_1'$ and $f_4'$ on $B$ form the backup slice $L_1 :: B :: B$. During layer restoring, $L_1$ can be restored by using any combination of primary and backup slices to achieve maximum parallelism.

5.4.3 Restoring Layers

The restoring process works in two phases: slice reconstruction and layer reconstruction. Considering the example in Figure 5.5, restoring works as follows:

According to $L_1$’s layer recipe, the restoring workers are D-servers $A$, $B$, and $C$. The node with the largest slice is picked as the restoring master, also called layer constructor ($A$ in
Chapter 5. Flexible High-Performance Deduplication for Docker Registries

Figure 5.6: Parallel streaming layer construction.

the example). Since $A$ is the restoring master it sends GET slice requests for the primary slices to $B$ and $C$. If a primary slice is missing, the master locates its corresponding backup slice and sends a GET slice request to the corresponding D-server.

After a GET slice request has been received, $B$’s and $C$’s slice constructors start rebuilding their primary slices and send them to $A$ as shown in Figure 5.6. Meanwhile, $A$ instructs its local slice constructor to restore its primary slice for $L1$. To construct a layer slice, a slice constructor first gets the associated slice recipe from the metadata database. The recipe is keyed by a combination of layer Id, host address and requested backup level, e.g., $L1 :: A :: P$. Based on the recipe, the slice constructor creates a slice tar file by concatenating each file header and the corresponding file contents; it then compresses the slice and passes it to the master. The master concatenates all the compressed slices into a single compressed layer tarball and sends it back to the client.

The layer restoration performance is critical to keep pull latencies low. Hence, Duphunter parallelizes slice reconstruction on a single node and avoids generating intermediate files on disk to reduce disk I/O.

5.4.4 Caching and Preconstructing Layers

Duphunter maintains a cache layer in both the primary and deduplication clusters to speedup pull requests. The primary cluster cache (in-memory prefetch cache) is to avoid disk I/O during layer retrievals while the deduplication cluster on-disk cache stores preconstructed layers, which are likely to be accessed in the future. Both caches are filled based on the user access patterns seen in §5.3.

Request prediction  To accurately predict layers that will be accessed in the future, Duphunter keeps track of image metadata and user access patterns in two data structures: $ILmap$ and $ULmap$. $ILmap$ maps an image to its containing layer set. $ULmap$ stores for each user the layers the user has accessed and the corresponding pull count. A user is uniquely identified by extracting the sender network address from the request.

When a GET manifest request $r$ is received, Duphunter first calculates a set of image layers
that have not been pulled by the user $r.addr$ by calculating the difference $S_{\Delta}$ between the image’s layer set and the user’s accessed layer set:

$$S_{\Delta} = ILmap[r.img] - ULmap[r.addr].$$

The layers in $S_{\Delta}$ are expected to be accessed soon.

Recall from §5.3.3 that some users *always* pull layers, no matter if the layers have been previously pulled. To detect such users, Duphunter maintains a repull probability $\gamma$ for each user. For a GET manifest request $r$ by a user $r.addr$, $\gamma$ is computed as

$$\gamma[r.addr] = \frac{\sum_{l \in RL} l.pullCount}{\sum_{l \in L} l.pullCount}$$

where $RL$ is the set of layers that the user has repulled before (i.e. with a pull count $> 1$) and $L$ is the set of all layers the user has ever pulled. Duphunter updates the pull counts every time it receives a GET layer request.

Duphunter compares the clients’ repull probability to a predefined threshold $\varepsilon$. If $\gamma[r.addr] > \varepsilon$, then Duphunter classifies the user as a repull user and computes the subset, $S_{\gamma}$, of layers from the requested image that have already been pulled by the user:

$$S_{\gamma} = ILmap[r.img] \cap ULmap[r.addr].$$

It then fetches the layers in $S_{\gamma}$ into the cache.

**Cache handling in tiered storage** The introduction of the two caches results in a 5-level 2-tier storage architecture of Duphunter as shown in Figure 5.7. Requests are passed through the tiers from top to bottom. Upon a GET layer request, Duphunter first determines the P-server(s) which is (are) responsible for the layer and searches the prefetch cache(s). If the layer is present the request will be served from the cache. Otherwise, the request will be served from the layer store.

If a GET layer request cannot be served from the primary cluster due to a failure of the corresponding P-server(s), the request will be forwarded to the deduplication cluster. In that case, Duphunter will first lookup the layer recipe. If not found, it means that the layer has not been fully deduplicated yet and Duphunter will serve the layer from one of the layer stage areas of the responsible D-servers. If the layer recipe is present, Duphunter will contact...
the restoring master to check, whether the layer is in its preconstruct cache. Otherwise, it will instruct the restoring master to rebuild the layer.

Both the prefetch and the preconstruct caches are write-through caches. When a layer is evicted, it is simply discarded since the layers are read-only. We use an Adaptive Replacement Cache (ARC) replacement policy [130], which keeps track of both the frequently and recently used layers and adapts to changing access patterns.

5.4.5 Discussion

The goal of Duphunter is to provide flexible deduplication modes to meet different space-saving and performance requirements and mitigate layer restore overhead. The above design of Duphunter mainly focuses on file-level deduplication and assumes layer replication.

To achieve a higher deduplication ratio, Duphunter can integrate with block-level deduplication. After removing redundant files, D-servers can further perform block-level deduplication only on unique files by using systems such as VDO [182] and Ceph [140]. However, higher deduplication ratios come with higher layer restoring overhead as the restoring latency for block-level deduplication is higher than that of file level as we show in §5.5. This is because to restore a layer, its containing files need to be first restored, which incurs extra overhead. Furthermore, when integrating with a global block-level deduplication scheme, the layer restoring overhead will be higher due to network communication. In this case, it’s beneficial to maintain a number of layer replicas on P-servers to maintain a good performance.

While Duphunter exploits existing replication schemes, it is not limited to those. If the registry is using erasure coding for reliability, Duphunter can integrate with the erasure coding algorithm to improve space efficiency. Specifically, after removing redundant files from layers, Duphunter can store unique files as erasure-coded chunks. While Duphunter can not make use of existing replicas to improve pull performance in this case, its preconstruct cache remains beneficial to mitigate high restoring overheads as shown in §5.5.

5.5 Evaluation

We answer two questions in the evaluation: how do deduplication modes impact the performance-redundancy trade-off and how effective are Duphunter’s caches.

5.5.1 Evaluation Setup

Testbed Our testbed consists of a 16-node cluster, each node equipped with 8 cores, 16 GB RAM, a 500 GB SSD, and a 10 Gbps NIC.
5.5. Evaluation

Table 5.2: Workload parameters.

<table>
<thead>
<tr>
<th>Trace</th>
<th>#GET Layer</th>
<th>#GET Manifest</th>
<th>#PUT Layer</th>
<th>#PUT Manifest</th>
<th>#Uniq. Layer</th>
<th>#Accessed Uniq. Dataset Size (GB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dal</td>
<td>6963</td>
<td>7561</td>
<td>453</td>
<td>23</td>
<td>1870</td>
<td>18</td>
</tr>
<tr>
<td>Fra</td>
<td>4117</td>
<td>10350</td>
<td>508</td>
<td>25</td>
<td>1012</td>
<td>9</td>
</tr>
<tr>
<td>Lon</td>
<td>2570</td>
<td>11808</td>
<td>582</td>
<td>40</td>
<td>1979</td>
<td>13</td>
</tr>
<tr>
<td>Syd</td>
<td>3382</td>
<td>11150</td>
<td>453</td>
<td>15</td>
<td>558</td>
<td>5</td>
</tr>
</tbody>
</table>

Dataset  We download 0.93 TB of popular Docker images (i.e. images with a pull count greater than 100) with 36,000 compressed layers, totalling 2 TB after decompression. The file-level deduplication ratio of the decompressed dataset is 2.1.

Workload generation  To evaluate how Duphunter performs with production registry workloads, we use the IBM Cloud Registry traces [48] which come from four production registry clusters (, , , and ) and span approximately 80 days. We use Docker registry trace player [48] to replay the first 15,000 requests from each workload as shown in Table 5.2. We modify the player to match requested layers in the IBM trace with real layers downloaded from Docker Hub based on the layer size. Consequently, each layer request pulls or pushes a real layer. For manifest requests, we generate random well-formed, manifest files.

In addition, our workload generator uses a proxy emulator to decide the server for each request. The proxy emulator uses consistent hashing [115] to distribute layers and manifests. It maintains a ring of registry servers and calculates a destination registry server for each push layer or manifest request by hashing its digest. For pull manifest requests, the proxy emulator maintains two consistent hashing rings, one for the P-servers, and another for the D-servers. By default, the proxy first queries the P-servers but if the requested P-server is not available, it pulls from the D-servers.

Schemes  We evaluate Duphunter’s deduplication ratio and performance using different deduplication and redundancy schemes. The base case considers 3-way layer replication and file-level deduplication. In that case, Duphunter provides five deduplication modes: B-mode 0, 1, 2, 3, and S-mode. Note that B-mode 0 deduplicates all layer replicas (denoted as global file-level deduplication with replication or $GF-R$) while B-mode 3 does not deduplicate any layer replicas.

To evaluate how Duphunter works with block-level deduplication, we integrate B-mode 0 with VDO. For each D-server, all unique files are stored on a local VDO device. Hence, in that mode Duphunter provides global file-level deduplication and local block-level deduplication.

\footnote{The original player generates random or zeroed data for layers.}
Table 5.3: Dedup. ratio vs. GET layer latency.

<table>
<thead>
<tr>
<th>Mode</th>
<th>Dedup. ratio</th>
<th>Performance improvement (P-servers)</th>
</tr>
</thead>
<tbody>
<tr>
<td>B-mode 1</td>
<td>1.5</td>
<td>1.6×</td>
</tr>
<tr>
<td>S-mode</td>
<td>1.3</td>
<td>2×</td>
</tr>
<tr>
<td>B-mode 2</td>
<td>1.2</td>
<td>2.6×</td>
</tr>
<tr>
<td>B-mode 3</td>
<td>1</td>
<td>2.8×</td>
</tr>
<tr>
<td>GF-R</td>
<td>Dedup ratio</td>
<td>Performance degradation (D-servers)</td>
</tr>
<tr>
<td></td>
<td>2.1</td>
<td>-1.03 ×</td>
</tr>
<tr>
<td>GF+LB-R</td>
<td>3.0</td>
<td>-2.87 ×</td>
</tr>
<tr>
<td>GB-EC</td>
<td>6.9</td>
<td>-6.37 ×</td>
</tr>
</tbody>
</table>

Figure 5.8: Cache hit ratio on P-servers with different cache algorithms.

We also evaluate Duphunter with an erasure coding policy instead of replication. We combine B-mode 0 with Ceph such that each D-server stores unique files on a Ceph erasure coding pool with global block-level deduplication enabled. We denote this scheme as GB-EC. We compare each scheme to Bolt [127] with 3-way replication as our baseline (No-dedup).

5.5.2 Deduplication Ratio vs. Performance

We first evaluate Duphunter’s performance/deduplication ratio tradeoff for all the above described deduplication schemes. For the replication scenarios, we use 3-way replication and for GB-EC, we use a (6,2) Reed Solomon code [157, 160]. Both replication and erasure coding policies can sustain the loss of two nodes. We use 300 clients spread across 10 nodes and measure the average GET layer latency across the four production workloads. Table 5.3 shows the results normalized to the baseline.
We see that all four performance modes of Duphunter (B-mode 1, 2, and 3, and S-mode) have better GET layer performance compared to No-dedup. B-mode 1 and 3 reduce the GET layer latency by 1.6× and 2.8×, respectively. This is because the prefetch cache hit ratio on P-servers is 0.98 and a high cache hit ratio significantly reduces disk accesses. B-mode 3 has the highest GET layer performance but does not provide any space savings since each layer in B-mode 3 has three full replicas. B-mode 1 and 2 maintain only one and two layer replicas for each layer, respectively. Hence, B-mode 1 has a lower performance improvement (i.e. 1.6×) than B-mode 2 (i.e. 2.6×), but has a higher deduplication ratio of 1.5×. S-mode lies between B-mode 1 and 2 in terms of the deduplication ratio and performance. This is because, in S-mode, popular layers have three layer replicas while cold layers only have a single replica.

Compared to the above four modes, B-mode 0 has the highest deduplication ratio because all layer replicas are deduplicated. Consequently, B-mode 0 adds overhead to GET layer requests compared to the baseline performance. As shown in Table 5.3, if file-level deduplication and 3-way replication are used, the deduplication ratio of B-mode 0 is 2.1 while the GET layer performance is 1.03× slower.

If block-level deduplication and block-level compression are used (GF+LB-R), the deduplication ratio increases to 3.0 while the GET layer performance decreases to 2.87×. This is because of the additional overhead added by restoring the layer’s files prior to restoring the actual layer. Compared to replication, erasure coding naturally reduces storage space. The deduplication ratio with erasure coding and block-level deduplication is the highest (i.e. 6.9). However, the GET layer performance decreases by 6.37× because to restore a layer, its containing files, which are split into data chunks and spread across different nodes, must first be restored.

Overall though, Duphunter, even in B-mode 0, significantly decreases the layer restoring overhead compared to the naive approaches shown in Table 5.1 in §5.3.2. For example, Duphunter B-mode 0 with VDO (the GF+LB-R scheme) has a GET layer latency only 2.87× slower than the baseline compared to a the VDO-only scheme which is 60× slower compared to the baseline.
Chapter 5. Flexible High-Performance Deduplication for Docker Registries

5.5.3 Cache Effectiveness

Next, we analyze Duphunter’s caching behavior. We first study the prefetch cache and then the preconstruct cache.

Prefetch cache. To understand how the prefetch cache improves the P-servers’ performance, we first show its hit ratio compared to two popular cache algorithms: LRU [20] and ARC [130]. Moreover, we compare Duphunter’s prefetch cache with another prefetch algorithm, which makes predictions based on PUT requests [48] (denotes as ARC+P-PUT). Both of these algorithms are implemented on ARC since ARC outperforms LRU. Duphunter’s prefetch algorithm, based on user behavior (UB), is denoted as ARC+P-UB. We vary the cache sizes from 5% to 15% of each workload’s unique dataset size. Figure 5.8 shows the results for the four production workloads (Dal, Syd, Lon, and Fra).

For a cache size of 5%, the hit ratios of LRU are only 0.59, 0.58, 0.27, and 0.10, respectively. ARC hit ratios are higher compared to LRU (e.g., $1.6 \times$ Lon) because after a user pulls a layer, the user is not likely to repull this layer in the future as it is locally available. Compared to
5.5. Evaluation

![Bar graph showing average wait time with different number of clients (Y-axis is log-scale).](image)

Figure 5.14: Average wait time with different number of clients (Y-axis is log-scale).

LRU, ARC maintains two lists, an LRU list and an LFU list, and adaptively balances them to increase the hit ratio.

ARC+P-PUT improves the ARC hit ratio by $1.9 \times$ for Lon. However, ARC+P-PUT only slightly improves the hit ratio for the other workloads. This is because ARC+P-PUT acts like a write cache which temporally holds recently uploaded layers and waits for the clients that have not yet pulled these layers to issue GET requests. This is not practical because the layer reuse time (i.e. interval between a PUT layer request and its subsequent GET layer request) is long. For example, the reuse time is 0.5 hr for Dal on average based on our observation. Moreover, ARC+P-PUT ignores the fact that some clients always repull layers. Duphunter’s ARC+P-UB achieves the highest hit ratio. For example, ARC+P-UB’s hit ratio for Dal is 0.89, resulting in a $4.2 \times$ improvement compared to ARC+P-PUT.

As shown in Figure 5.8, the hit ratio increases as the cache size increases. For example, when cache size increases from 5% to 15%, the hit ratio for ARC under workload Lon increases from 0.44 to 0.6. ARC+P-UB achieves the highest hit ratio of 0.96 for a cache size of 15% under workload Lon. Overall, this shows that by exploiting user behavior ARC+P-UB can achieve high hit ratios, even for smaller cache sizes.

Figure 5.9 shows the 99th percentile of GET request latencies for P-servers with different cache algorithms. The GET layer latency decreases with higher hit ratios. For example, when the cache size increases from 5% to 15%, the 99th percentile latencies decrease from 0.19 s to 0.15 s for Duphunter’s ARC+P-UB under workload Dal and the cache hit ratio increases from 0.8 to 0.92. Moreover, when the cache size is only 5%, ARC+P-UB significantly outperforms the other 3 caching algorithms. For example, ARC+P-UB reduces latency by $1.4 \times$ compared to LRU for workload Fra. Overall, ARC+P-UB can largely improve GET layer performance for P-servers with a small cache size.

**Preconstruct cache.** For the preconstruct cache to be effective, layer restoring must be fast enough to complete within the time window between the GET manifest and GET layer request.
Preconstruct cache impact. To understand how the preconstruct cache improves D-servers’ GET layer performance, we first show its hit ratio on D-servers with three deduplication schemes (GF-R, GF+LB-R, and GF-EC). The cache size is set to 10% of the unique dataset.

Figure 5.11 shows the preconstruct cache hit ratio breakdown. **Hit** means the requested layer is present in the cache while **Wait** means the requested layer is in the process of preconstruction and the request needs to wait until the construction process finishes. **Miss** means the requested layer is neither present in the cache nor in the process of preconstruction. As shown in the figure, GF-R has the highest hit ratio, e.g., 0.98 for the Dal workload. Correspondingly, GF-R also has the lowest wait and miss ratios because it has the lowest restoring latency and a majority of the layers can be preconstructed on time.

Note that the miss ratio of the preconstruct cache is slightly lower than that of the prefetch cache across all traces. This is because we use an in-memory buffer to hold the layer archives that are in the process of construction to avoid disk I/O. After preconstruction is done, the layers are flushed to the on-disk preconstruct cache. In this case, many requests can be served directly from the buffer and consequently, layer preconstruction does not immediately trigger cache eviction like layer prefetching. The preconstruct cache eviction is delayed till the layer preconstruction finishes.

GF+LB-R shows a slightly higher wait ratio than GF-R. For example, the wait ratios for GF-R and GF+LB-R are 0.04 and 0.06, respectively under workload Syd. This is because the layer restoring latency of GF+LB-R is slightly higher than GF-R. GB-EC’s wait ratio is the highest. Under workload Syd, 39% of GET layer requests are waiting for GB-EC as layers cannot be preconstructed on time.

Figure 5.12 shows the corresponding average GET layer latencies of D-servers compared to No-dedup. GF-R and GF+LB-R increase the latency by 1.04× and 3.1×, respectively, while GB-EC adds a 5× increase. This is due to GB-EC’s high wait ratios.

Scalability. To analyze the scalability of the preconstruct cache under higher load, we increase the number of concurrent clients sending GET layer requests, and measure the request wait ratio (Figure 5.13) and the average wait time (Figure 5.14).

Under workload Fra and Syd, the wait ratio for GB-EC increases dramatically with the number of concurrent clients. For example, the wait ratio increases from 15% to 28% as the number of concurrent clients increases from 50 to 300. This is because the layer restore latency for GB-EC is higher and with more concurrent client requests, more requested layers cannot be preconstructed on time. Under workload Lon and Dal, the wait ratio for GB-EC remains stable. This is because the client requests are highly skewed. A small number of clients issue the majority of GET layer requests. Correspondingly, GB-EC also has the highest wait time. Under workload Fra and Syd, the average wait time increases from 0.6 s to 1.1 s and 0.4 s to 1.4 s respectively as the number of clients increases from 50 to 300 for GB-EC.
Although some layers cannot be preconstructed before the GET layer requests arrive, the preconstruct cache can still reduce the overhead because layer construction starts prior to the arrival of the GET requests. This is shown by the fact that the wait times are significantly lower than the layer construction times. For GF-R and GF+LB-R, the average wait times are only 0.001 s and 0.003 s, respectively under workload Dal. When the number of concurrent clients increases, the average wait time of GF-R and GF+LB-R remains low. This means that the majority of layers can be preconstructed on time for both GF-R and GF+LB-R, and the layers that cannot be preconstructed on time do not incur high overhead.

Layer restoring performance. To understand the layer restoring overhead, we disable the preconstruct cache and measure the average GET layer latency when a layer needs to be restored on D-servers. We evaluate GB-EC, GB+LB-R, and GF-R and compare it to No-dedup.

We break down the average reconstruction latency into its individual steps. The steps in layer reconstruction include looking up the layer recipe, fetching and concatenating slices, and transferring the layer. Fetching and concatenating slices in itself involves slice recipe lookup, slice packing, slice compression, and slice transfer. No-dedup contains three steps: layer metadata lookup, layer loading from disk to memory, and layer transfer.

As shown in Figure 5.10, GF-R has the lowest layer restoring overhead compared to GF+LB-R and GB-EC. It takes 0.44 s to rebuild a layer tarball for GF-R. Compared to the No-Dedup scheme, the GET layer latency of GF-R increases by 3.1×. Half of the GET layer latency is spent on slice concatenation. This is because slice concatenation involves writing each slice into a compressed tar archive, which is done sequentially. Slice packing and compression are faster, 0.07 s and 0.05 s, respectively, because slices are smaller and evenly distributed on different D-servers.

For the GF+LB-R scheme, it takes 0.55 s to rebuild a layer. Compared to GF-R, adding local block-level deduplication increases the overall overhead by up to 1.4× due to more expensive slice packing. It takes 0.18 s to pack a slice into an archive, 2.7× higher than GF-R’s slice packing latency as reading files from the local VDO device requires an additional file restoring process.

The GB-EC scheme has the highest layer restoring overhead. The bottleneck is again slice packing which takes 5 s. This is because each file is split into four data chunks, distributed on different D-servers, and deduplicated. To pack a slice, each involved file needs to be reconstructed from different D-servers and then written to a slice archive, which incurs considerable overhead.
5.6 Chapter summary

We presented Duphunter, a new Docker registry architecture that provides flexible and high performance deduplication for container images and reduces storage utilization. Duphunter provides multiple configurable deduplication modes to meet different space saving and performance requirements. Additionally, it parallelizes layer reconstruction locally and across the cluster to further mitigate overheads. Moreover, by exploiting knowledge on the application domain, it introduces a two-tier storage hierarchy with a novel layer prefetch/pre-construct cache algorithm based on user access patterns. Duphunter’s prefetch cache can improve \texttt{GET} latencies by up to $2.8 \times$ while the preconstruct cache can reduce the restore overhead by up to $20.9 \times$ compared to the state-of-the-art.
Chapter 6

Adaptive Wear Balancer for Flash Clusters

6.1 Introduction

Flash memory has emerged as a viable storage alternative for mobile computing devices due to its high throughput, persistence, and lower power consumption. The development of commodity flash devices such as solid state drives (SSDs) has also expanded flash memory’s role in enterprise storage servers. All-flash or disk-free server storage systems (e.g., FlashStore [85] and Analyzethis [165]) are being developed. Flash-based storage servers that can play a significant role in accelerating application performance are clustered together and managed as a single entity for high reliability and availability in many distributed storage platforms such as FAWN [32], BlueDBM [112], QuickSAN [67] and CORFU [52].

Unlike magnetic disk drives, flash devices read and write data at the granularity of pages but erase data in units of a block. SSDs typically provide a flash translation layer (FTL) within the device to manage garbage collection (GC). If some valid pages are physically located in a block (called victim block) that has some invalid pages that need recycling, GC will first copy the valid data to a free page and then erase the victim block to make the block available for new writes. Consequently, a write operation can lead to multiple writes, resulting in write amplification. In this case, GC is time consuming (relative to read/write speeds) and also affects device endurance, as the number of sustainable erasures (P/E cycles) of a given block is limited.

To improve flash endurance and lifetime, FTL uses wear leveling (WL) to evenly wear all the flash blocks within a flash device so that no block will be worn out faster than others. However, the I/O workloads served by the flash based storage servers are imbalanced [82, 146, 147, 148, 149, 150, 166, 183], which incur wear imbalance among flash servers. For example, Facebook’s distributed key-value (KV) store workload analysis [39, 41, 43, 44, 45, 51] reports
high access skew and time varying workload patterns. The flash devices associated with heaviy loaded servers serve more writes and perform more GCs, and thus wear out faster than others in a deployment. The maintenance of SSD devices raises many concerns. For instance, any maintenance required by the storage devices may require taking the entire flash server offline. This not only incurs administrative cost but also performance degradation, especially when SSDs are crucial, e.g., for burst buffer I/O nodes performance [25]. Non-uniform I/O workload, coupled with wear imbalance, also impact write performance because frequent GCs in flash servers with high utilization cause overall system slow down [33, 42, 46, 47, 63, 69, 78]. Moreover, wear imbalance worsens when fault tolerant or data redundancy schemes such as replication (REP) or erasure coding [65, 141, 188] (EC) are applied in a flash cluster. This is mainly because storing extra redundant data generates more writes, which in turn severely impact flash endurance.

**Motivational Study.** To quantify the impact of non-uniform write intensity on the SSD erasure count, we built a distributed flash-based KV store that maps data to a 50-node cluster using consistent hashing [114]. Each node stores data locally in an SSD device that is simulated using FlashSim [117]. We applied two kinds of redundancy policies separately: REP with replication level \( r = 3 \), and EC with RS (6,4) encoding [15]. We measure total erasure count under YCSB workload [81] with Zipf-like access pattern (\( YCSB - zipf \)), and two block-level traces from MSR-Cambridge data center servers [139], namely, \( prn_0 \) and \( proj_0 \).

Figures 6.1(a) and 6.1(b) show the sorted erasure count distribution under REP and EC, respectively. The largest erasure count is \( 4 \times \) more than the smallest erasure count for \( proj_0 \), and \( 3 \times \) for both \( prn_0 \) and \( YCSB - zipf \) under REP. Under EC, the largest erasure count is \( 12 \times \) more than the smallest erasure count for \( YCSB - zipf \), and \( 3 \times \)
for the other two workloads. These results show that the erasure counts are highly skewed among flash servers both under REP and EC schemes. Moreover, REP experiences almost $2 \times$ more erasure counts than EC.

To address the challenge of balancing wear across flash servers, many researchers take inspiration from data migration \cite{74, 75, 144, 151}. For example, EDM \cite{144} is a data migration-based wear balancing algorithm. It moves data from the flash servers with higher erasure count to the ones with lower erasure count for balancing the wear speeds. However, the extra writes generated by data migration create additional overhead, which incurs considerable write amplification overhead and consequently causes more GCs and significant extra erasure count to the flash cluster \cite{37, 38, 40, 121, 122}. Moreover, the redundancy policies are completely ignored during wear balancing. However, we observed that the redundancy policies can provide useful information that can be leveraged to improve wear balance and flash lifetime, as well as performance.

**Contributions.** To solve the problems of multi-server wear imbalance, we propose a practical and efficient global wear balancing technique, Chameleon. Chameleon quickly detects the presence of erasure imbalance in a flash cluster. The goal is to balance the erasure count and improve both lifetime and performance of the flash cluster.

Specifically, this chapter makes the following contributions:

- We exploit two redundancy policies–REP and EC–to help improve wear balance and flash lifetime, while also improving performance.

- We take advantages of the out-of-place update feature of flash memory by directly offloading the writes/updates across flash servers instead of moving data across flash servers to mitigate extra-wear cost, which includes late replicating (Late REP), late encoding (Late EC), and endurance aware write offloading (EWO).

- We provide two adaptive wear-balancing algorithms, redundancy policy transition (ARPT) and Hot/Cold data swapping (HCDS), coupled with write offloading and redundancy policies to balance the erasure count and improve both lifetime and performance of a flash cluster.

- We integrate our Chameleon emulator with a distributed flash-based KV store application. Emulation results on real-world workloads show that Chameleon outperforms the state-of-the-art data migration based wear balancing technique, reducing up to 81% wear variance while improving the write performance by up to 33%.
6.2 Background and Motivation

**Flash endurance.** A large body of work has examined flash endurance [56, 73, 110, 169]. Techniques such as log-structured caching [169], inclusion of combining multiple bad blocks into virtual healthy blocks [110] have been explored to improve the lifetime of flash devices. These works are orthogonal and complementary to Chameleon.

**Intradisk wear leveling.** Dynamic [113, 129] techniques aim to achieve a good wear evenness while keeping the overhead low. Similarly, static wear leveling techniques [71, 109, 110, 137] move cold data to the blocks with higher erasure counts, thereby improving the even spread of wear. Chameleon leverages such approaches for extending the lifetime of individual SSDs in its target distributed setting.

**Interdisk wear leveling.** Application of SSD arrays in enterprise data-intensive applications is growing. In such an environment, we have observed significant variance in number of writes and merge operations that are sent to individual SSDs. Recent work [98] manages EC stripes to increase reliability and operational lifetime of such flash memory-based storage systems, and uses a log-structured approach that does not need explicit wear balancing as data is appended and not updated in place. In contrast, EDM [145] also targets SSD arrays but use data migration to achieve wear balance across the SSDs in the array. SWANS [186] dynamically monitors the variance of write intensity across the array and redistributes writes based only on the number of writes that an SSD has received to prolong the SSD arrays’ service life. These methods share with Chameleon the goal of wear leveling across an SSD array, however unlike them Chameleon considers the role of redundant policies at various storage hierarchy and their impact on overall wear balancing.

**Distributed flash storage systems.** FAWN [32] uses small amounts of local flash storage across a number of low-power resource-constrained nodes to enable a consistent and replicated key-value storage system. CORFU [52] extends the local log-structured design by organizing the entire cluster of SSDs as a global shared log. Both of these systems utilize homogeneous nodes and replication for high availability. Other works [35, 76, 77, 120] focuses on tiered storage to reduce the load on flash devices. Similarly, [34, 36] use data partitioning to evenly distribute load. In contrast, Chameleon focuses on EC storage solutions, which offer higher storage efficiency and exploits the interactions between the storage hierarchy to improve overall flash lifetime in flash-based clusters.
6.3 System Design

Chameleon is aimed to address challenges arising from modern I/O workloads that exhibit high skewness across distributed flash servers. If a flash cluster does not implement server-level wear balancing, a flash server with high write intensity can have huge erasure counts and be worn out, while other flash servers are underutilized. This uneven use will trigger degraded performance, and eventual maintenance and downtime that would affect overall system performance and availability.

Chameleon is designed to balance the wear among different flash servers with the goal of: (1) reducing the unnecessary administration cost of replacing worn-out or failed flash devices; (2) improving the average lifespan of all flash devices; and (3) improving performance especially for write operations.

We first describe the framework of Chameleon in Section 6.3.1 followed by two adaptive wear-balancing algorithms detailed in Section 6.3.2. The mapping table is discussed in Section 6.3.3.
6.3.1 Overview of Chameleon architecture

Figure 6.2(a) shows an overview of Chameleon architecture comprising four modules: flash monitor, wear balancer, mapping table, and client library. Instances of Flash monitor and wear balancer are distributed across the flash-based storage servers (called flash servers). These components monitor and balance the wear of the whole cluster. Chameleon keeps track of objects related metadata (e.g., location and popularity) and stores the metadata in a distributed database as a distributed mapping table. The use of distributed database helps Chameleon scale as needed. The Client library provides a basic interface for users to read and write data to flash servers. Read or write requests are sent to Chameleon instances that determine the location of the flash devices for serving the requests.

Flash monitor monitors the statistics (i.e., erasure count and flash space utilization) of flash devices, and sends them to the wear balancer. In our current implementation, Chameleon assumes the host has full control over garbage collection (GC) as provided by open-channel SSDs [21]. The argument behind moving GC management from the flash to the host is that the host has better overall knowledge (e.g., total erasure count) that offers both better performance and more optimization opportunities, compared to the individual device FTL [124]. We focus on open-channel SSDs, as more and more of such devices are being introduced to the market, e.g., LightNVM [55]. Thus, by focusing on open-channel SSDs we can optimize both the current available components, as well as a growing number of components that will become available in the near future.

Wear balancer is responsible for balancing the wear of the whole cluster. Balancer has two major components as shown in Figure 6.2(b): (1) An adaptive redundancy policy transition (ARPT) module that dynamically converts data redundancy and adapts to workload changes for balancing the wear while ensuring good performance and low erasure overhead. (2) A data swapping module that swaps data between the servers with higher erasure counts and servers with lower erasure counts to further improve wear balance.

Mapping table keeps track of the updates made to the metadata when data object’s addresses and redundancy policies are changed during the balancing process. The table stores objects’ metadata, such as object’s state, popularity, and location. It also keeps track of the object access history (i.e., popularity) to facilitate wear balancing. Mapping table is kept in a distributed database to avoid memory overhead.
6.3.2 Adaptive wear balancing algorithms

In this section, we describe two wear balancing algorithms used by Chameleon: (1) Adaptive redundancy policy transition (ARPT) algorithm, and (2) Hot/cold data swapping (HCDS) algorithm.

ARPT adopts a hot/cold data segregation approach by leveraging: (1) REP to store a small fraction of mostly frequently updated data (write hot data) to provide overall low I/O latency for the system; and (2) EC to encode all the remaining relatively cold data to realize a low erasure overhead. Moreover, ARPT dynamically adapts to workload changes by using late-REP or late-EC (§6.3.2) to switch data state between two redundancy schemes and remap data for balancing the wear of whole cluster with low overhead. Furthermore, HCDS is used to swap hot and cold data between servers with higher erasure counts with the ones with lower erasure counts to further improve the wear balance.

---

**Algorithm 1:** User-based caching algorithm.

**Input:** \( \sigma \): cluster wear variance, \( \sigma_{ARPT} \): wear variance threshold, \( \ell_{hot} \): object popularity threshold.

**Require:** \( \sigma > \sigma_{ARPT} \)

**Ensure:** \( \sigma \leq \sigma_{ARPT} \)

1. // Step 1, Screen candidates from each server
2. for each object \( \text{obj}_i \) that is in flash cluster do
3.   if \( \text{obj}\_\text{popularity}(\text{obj}_i) \geq \ell_{hot} \) 
4.     //Convert its’redundancy scheme to late-REP
5.     \text{Convert.object.state}(_\text{obj}_i, \text{late-REP})
6.   end if
7.   if \( \text{obj}\_\text{popularity}(\text{obj}_i) < \ell_{hot} \) 
8.     //Convert its’redundancy scheme to late-EC
9.     \text{Convert.object.state}(_\text{obj}_i, \text{late-EC})
10. end if
11. end for
12. // Step 2, Rearrange candidates among nodes
13. while \( \sigma > \sigma_{ARPT} \) do
14.   \( X(x_1, x_2, x_3) \) \( \triangleright \) extract servers with MIN erasure counts
15.   \( Y(y_1, y_2, y_3, y_4, y_5, y_6) \) \( \triangleright \) extract servers with MAX erasure counts
16.   \( \text{obj}_i \) \( \triangleright \) Get_hottest_candidate (from step 1)
17.   \( \text{obj}_j \) \( \triangleright \) Get_coldes_candidate (from step 1)
18.   Map_object_to(\text{obj}_i, X)
19.   Map_object_to(\text{obj}_j, Y)
20. \( \sigma \) \( \triangleright \) Estimate wear variance
21. end while
Adaptive redundancy policy transition (ARPT). Chameleon tracks erasure counts to decide when the wear balancing process should be triggered. We define the wear variance $\sigma$ as the standard deviation of erasure counts. If the system develops significant wear imbalance—indicated by $\sigma > \sigma_{ARPT}$, where $\sigma_{ARPT}$ is a preset wear variance threshold (Table 6.1)—the balancing process is triggered periodically until the wear variance drops below the threshold.

Moreover, Chameleon also records the object write heat changes. Each object is classified as either hot or cold based on their write heat changes and the object’s state switches between REP and EC.

Chameleon performs a periodic scan through all the replicated data for “cooled down” data and convert such data’s redundancy policy from REP to EC, a process denoted as **downgrade**. Similarly, encoded data is also scanned for new hot data and these new hot data’s redundancy policy is switched from EC to REP, denoted as **upgrade**.

**Late-EC & Late-REP.** The additional erasure count caused during downgrade/upgrade operations is nontrivial. The downgrade operation requires network transfers of the replicated objects from different locations to encode them into RS code, along with invalidation of the old replicated objects. Upgrade operation needs to retrieve the data stripes from different locations to $k$-way replicate them and invalidate the old stripes. Both downgrade and upgrade operations will incur network overhead and extra erasure cycles. To mitigate this, Chameleon implements two additional optimizations, late-REP or late-EC, to support downgrade/upgrade with low overhead. Here, the conversion due to upgrade and downgrade are delayed until the next update, which not only reduces conversion overhead but also avoids unnecessary conversions, such as, a downgrade followed by an upgrade for the same data.

Downgrade/upgrade operations are delayed as long as the wear variance remains tolerable. The late policies trades-off the probability of wear imbalance with network traffic overhead. To do this, we exploit the out-of-place update feature of flash memory by delaying the redundancy policy transition until clients issue the write/update requests to the objects whose redundancy policies need to be converted. Then, we directly convert the requested data into the desired redundancy policy state (replicas or EC stripes) and re-distributes them to their respective destinations. Thus, the network traffic overhead can be reduced and the number of extra writes during redundancy transition process are mitigated.

As shown in Figure 6.2(b), we define two kinds of states for objects: redundancy states which contain REP and EC, and intermediate states that contain late REP, late EC, REP-EWO, and EC-EWO (detailed in 6.3.2). Figure 6.2(b) shows the redundancy policy transition of an object. As the write heat of an object increases, the object either stays in REP state or is converted from EC to late REP state by ARPT. The object stays in the late REP state until the next write/update is received and the state is changed to REP. Similarly, if the write heat of an object decreases, its state either stays in EC or is converted to late EC state.
6.3. System Design

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>REP</td>
<td>Replication</td>
</tr>
<tr>
<td>EC</td>
<td>Erasure coding</td>
</tr>
<tr>
<td>ARPT</td>
<td>Adaptive redundancy policy transition</td>
</tr>
<tr>
<td>HCDS</td>
<td>Hot cold data swapping</td>
</tr>
<tr>
<td>EWO</td>
<td>Endurance aware write offloading</td>
</tr>
<tr>
<td>Downgrade</td>
<td>Conversion from REP to EC</td>
</tr>
<tr>
<td>Upgrade</td>
<td>Conversion from EC to REP</td>
</tr>
<tr>
<td>Late-REP</td>
<td>Late replicating</td>
</tr>
<tr>
<td>Late-EC</td>
<td>Late erasure coding</td>
</tr>
<tr>
<td>$\sigma$</td>
<td>Standard deviation of erasure counts</td>
</tr>
<tr>
<td>$\sigma_{ARPT}$</td>
<td>Wear variance threshold that triggers ARPT</td>
</tr>
<tr>
<td>$\ell_{hot}$</td>
<td>Popularity threshold</td>
</tr>
<tr>
<td>$w_j$</td>
<td>Number of writes to the object during epoch $j$.</td>
</tr>
<tr>
<td>$p_k$</td>
<td>Write heat of the object at the end of epoch $k$</td>
</tr>
<tr>
<td>$\mu$</td>
<td>Utilization of a victim block that needs to be cleaned</td>
</tr>
<tr>
<td>$B_p$</td>
<td>Number of pages per block</td>
</tr>
<tr>
<td>$W_t$</td>
<td>Number of page writes during a certain epoch $t$.</td>
</tr>
<tr>
<td>$E_t$</td>
<td>Block erasure counts during epoch $t$.</td>
</tr>
<tr>
<td>$\sigma_{HCDS}$</td>
<td>Wear variance threshold that triggers HCDS</td>
</tr>
</tbody>
</table>

if the current state is not EC. The state will be eventually converted to an encoded state upon next write/update.

Specifically, if object $obj_i$’s popularity is greater than a predefined threshold ($\ell_{hot}$) denoted as hot object, and its state is neither REP nor late-REP, $obj_i$’s state will be converted to late-REP. In contrast, if object $obj_i$’s popularity is smaller than $\ell_{hot}$ denoted as cold object, and its state is neither EC nor late-EC, the state will be converted to late-EC. Here, object popularity can be calculated by using Equation 6.1.

We use an exponential decay function [144] to record the write heat of an object. For a given object $i$, the time duration from the time when the object $i$ is created to the present time is split into $k + 1$ epochs, epoch 0, $\ldots$, epoch $k$. We define the popularity of each object as follows:

$$p_k = \sum_{j=0}^{k} \frac{w_j}{2^{k-j}},$$  \hspace{1cm} (6.1)

where $w_j$ denotes the number of writes that access the object during an epoch $j$. $p_k$ denotes the write heat of the object at the end of epoch $k$.  

Remapping. A key challenge is to determine where to store the converted replicas or EC stripes after redundancy transition to ensure a good wear balance across different flash servers. The wear balancing process uses an effective endurance-aware greedy algorithm. As shown in lines 1 to 11 of Algorithm 1, Chameleon first screens candidates whose popularity state changes from hot to cold by sorting objects based on their popularity. During upgrade, Chameleon’s greedy algorithm iteratively re-distributes the $k$ (where $k = 3$) replicas of hottest candidate object to the flash servers with the lowest erasure count as shown in lines 11 to 21, the replicas of $\text{obj}_i$ are mapped to server array $X(x_1, x_2, x_3)$. While during downgrade, the $n$ (where $n = 6$) stripes of coldest candidate object are remapped to the flash servers with the highest erasure count as shown in lines 11 to 21, the stripes of $\text{obj}_j$ are mapped to server array $Y(y_1, y_2, y_3, y_4, y_5, y_6)$.

To estimate the erasure count caused by a specific number of writes, we first define the erasure cost for flash memory as $1 - \mu$ according to [72, 116], where $\mu$ is the utilization of a victim block that needs to be cleaned during the GC process. That is, the erasure cost is the amount of valid pages $\mu$ that need to be rewritten per victim block of new space claimed $(1-\mu)$. Let $B_p$ be the number of pages per block and $W_t$ be the number of page writes during a certain epoch $t$. Then, after GC starts, the approximation for block erasure counts caused by $W_t$ page writes during epoch $t$ is:

$$E_t = \frac{W_t}{B_p \times (1 - \mu)} \quad (6.2)$$

At the end of each iteration, we estimate the new cluster wear variance $\sigma$. If $\sigma <= \sigma_{ARPT}$, ARPT will stop the iteration. To estimate the new $\sigma$, we first estimate the new erasure count of each server $x$ in array $X$ after re-mapping $\text{obj}_i$, as $E_x$: $E_x = E_x + E(\text{obj}_i)$, where $E(\text{obj}_i)$ can be calculated by using Equation 6.2. While the new erasure count of each server $y$ in array $Y$ after re-mapping $\text{obj}_j$ can be estimated as $E_y$: $E_y = E_y + E(\text{obj}_j)$, where $E(\text{obj}_j)$ can be calculated by using Equation 6.2.

Hot/cold data swapping (HCDS). To further improve wear balance, Chameleon uses data swapping to exchange the storage location of hot replicas and cold EC stripes. As shown in Algorithm 2, Chameleon first selects two servers, server $x$ with highest erasure cycles and server $y$ with lowest erasure cycles. Then, the coldest object $\text{obj}_i$ from $x$ and the hottest objects $\text{obj}_j$ from server $y$ are exchanged until their erasure count difference ($\sigma$) is less than a preset threshold $\sigma_{HCDS}$ (Table 6.1).

After we map $\text{obj}_i$ to server $y$, we estimate its new erasure count as $E_y$: $E_y = E_y + E(\text{obj}_i)$, where $E(\text{obj}_i)$ can be calculated by using Equation 6.2. Similarly, the new erasure count of server $x$ is given by $E_x$: $E_x = E_x + E(\text{obj}_j)$). The data swapping process stops when the predefined erasure variance threshold $\sigma <= \sigma_{HCDS}$ is met.
6.3. System Design

Algorithm 2: Hot/cold data swapping.

Require: $\sigma > \sigma_{HCDS}$

Ensure: $\sigma \leq \sigma_{HCDS}$

1: while $\sigma > \sigma_{HCDS}$ do
2:   $x \triangleright$ extract server with max erase cycles
3:   $y \triangleright$ extract server with min erase cycles
4:   $obj_i \triangleright$ Get_hottest_candidate from $x$
5:   $obj_j \triangleright$ Get_coldest_candidate from $y$
6:   Map(object_to($obj_i$, $y$)
7:   Map(object_to($obj_j$, $x$)
8:   $\sigma \triangleright$ Estimate wear variance
9: end while

Endurance-aware write offloading (EWO) To minimize the network traffic and wear balancing overhead, Chameleon offloads the writes/updates to the replicas or stripes to their destination servers instead of migrating data via bulk data transfer through the network. Incoming writes/updates trigger the data mapping change eventually.

As shown in Figure 6.2(b), the hot/cold data swapping module swaps the hot and cold data from the highly loaded (in terms of erasures) server to lightly loaded server using endurance-aware write offloading. There are two intermediate states: REP-EWO and EC-EWO. As seen in the figure, a replica is selected to be exchanged with a EC-stripe. It will first be converted to the intermediate state REP-EWO until the next write/update is received and the state is changed to REP. Similarly, if an encoded stripe is chosen to swap with a replica, its state is converted to EC-EWO state. The state will be eventually converted to the EC state upon next write/update.

Ideally, EWO can offload all the candidate replicas or stripes to their destination servers. However, there is rarely accessed cold data that has not been accessed for a long period. For such data, we trade off network traffic overhead with better wear balancing by migrating the cold stripes to the destination servers.

6.3.3 Mapping table

The data mapping relationship from client to flash servers is changed dynamically. Different versions of the same data can be stored on multiple different locations because of redundancy scheme conversion and data swapping. Chameleon must ensure that reads always go to the location holding the latest version of the data. To ensure read correctness, Chameleon uses a mapping table to efficiently manage the objects whose redundancy scheme and addresses have been changed during wear balancing process as shown in Figure 6.3.
Source/Destination server array. Chameleon provides two levels of indirection for locating servers so as to reduce metadata overhead while maintaining read and write correctness. For a given Objᵢ, the first level indirection indicates the former data host, i.e., the source server array for intermediate states, e.g., late-EC, late-REP, EC-EWO, or REP-EWO state. The second level indirection represents the object’s destination server array for intermediate states or its current host for redundancy states, e.g., REP or EC state.

Assume Chameleon monitors the cluster wear variance in a fixed time interval, denoted as epoch. As shown in figure 6.3, during epoch 0, Obj₀ is selected for redundancy policy transition from EC to REP. Obj₀’s state is late-REP, which means that Chameleon will wait until an update/write request accesses Obj₀. When a write request accesses Obj₀, Chameleon directly replicates the request data and distributes its R replicas on the associated destination servers denoted as array D_node_arr and then changes Obj₀’s redundancy state to REP.

For a read request, if the requested object’s state is an intermediate state, e.g., late-EC, late-REP, EC-EWO, or REP-EWO state, Chameleon sends the request to the object’s source server. The source server is denoted as the array S_Node_arr, and holds the latest version of the data as shown in Figure 6.3. Otherwise, read requests will be sent to the object’s destination servers.
Compaction. As mentioned before, to reduce network traffic during the wear balancing process, Chameleon uses late-EC/REP, and EWO techniques to make a compromise between network traffic and the risk of temporary wear imbalance by waiting for an update request to the state change object. However, this wait can be for a very long time especially for cold data. Moreover, even for a hot object, there may not be an update request to such an object during an epoch as workloads are unpredictable.

As shown in Figure 6.3, a hot Obj0 is selected to convert its redundancy scheme from EC to REP during epoch 0, but until epoch 3, there is still no updates to Obj0. So in the epoch 4, Chameleon classifies the object as cold data and converts its redundancy policy from REP to EC. Chameleon creates a metadata object with version 4 for Obj0, and appends the metadata to Obj0’s epoch log. In this case, Chameleon can keep track of each converted objects’ state/location changes for failure recovery. However, epoch log would incur considerable memory overhead since epoch log increases with number of wear balancing process and the amount of involved objects.

Chameleon uses compaction to combine epoch log for each remapped object to reduce memory overhead. As shown in Figure 6.3, the metadata object associated with Obj0 is updated from epoch version 0 to epoch version 4. The object’s state is marked as EC since till epoch 3, there is no update to convert the state from late EC to REP. This means that Obj0 is still encoded as stripes stored on its source destination array S_Node_arr. Thus, in epoch 4, the object’s source destination array S_Node_arr. becomes its destination array as shown in Figure 6.3.

Consequently, Chameleon only maintains a single updated metadata object for the current epoch version, which not only ensures the correctness of R/W requests but also can mitigate metadata overhead.

6.4 Evaluation

6.4.1 Implementation

We have implemented a prototype emulator of Chameleon using 16k lines of C++ code. We built a KV-store from scratch as a test application. We map data to servers by using a consistent hash-based data distribution algorithm that distributes data evenly across participating servers [114]. The hash function used in our experiments is FVN-a1 [12]. Each trace record maps to a logical object, which corresponds to a unique object ID calculated by using the consistent hash function. The logical object is then stored in the appropriate server by consulting the consistent hash table. We use ISA-L [15] for encoding and decoding operations. The Intel ISA-L library provides a highly optimized implementation of Reed-Solomon codes that significantly decreases the time taken for encoding and decoding operations. Specifically, we implemented our Chameleon as follows:
Table 6.2: SSD parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Page size</td>
<td>4KB</td>
</tr>
<tr>
<td>Block size</td>
<td>256KB</td>
</tr>
<tr>
<td>Read latency</td>
<td>25us</td>
</tr>
<tr>
<td>Write latency</td>
<td>200us</td>
</tr>
<tr>
<td>Erase latency</td>
<td>1.5ms</td>
</tr>
<tr>
<td>Over-provisioned space</td>
<td>15%</td>
</tr>
</tbody>
</table>

**Flash server and flash cluster.** We emulate a large flash cluster by running multiple instances of our SSD simulator as flash server nodes. We use Flashsim [117] to simulate the SSD behavior as Flashsim can accurately show the block erase cycles. For all of our tests, we use an evaluation testbed with 50 flash server nodes. Each flash server node is equipped with one SSD that is simulated by FlashSim. To improve the performance especially write performance, we built a local log on top of SSD simulator. All the writes are appended to tail of the log. Table 6.2 summarizes the parameters that are commonly used to simulate SSD.

**Flash monitor** runs on each flash server, monitors the statistics of SSDs, and sends them to the wear balancer. We modified Flashsim by adding a flash statistics collector to the code. For connectivity between flash server nodes, we integrate Google Protocol Buffer [23] in Flashsim to facilitate communication, such as protocol parsing and messaging.

**Wear balancer and mapping table** are also implemented along with flash monitor on each flash server. We integrated ZooKeeper [103] in our KV-store as a distributed coordination service. One flash server is chosen as a coordinator. The wear balancer running on the coordinator node gathers statistics of each flash server, such as the flash space utilization and erasure count by exchanging the heartbeat messages with the flash monitor running on each flash server. We installed MySQL on the flash cluster as a metadata service for storing the mapping table. Before performing wear balancing, the balancer running on the coordinator node first requests object popularity statistics from the mapping table. After wear balancing, the coordinator updates the metadata changes related to the remapped objects to the mapping table.

**Client library** provides a basic API to read/write the data to flash cluster and to choose between REP or EC as initial redundancy policy. For EC, the data is split into several data stripes and encoded with few parity stripes. Throughout our evaluation, we use RS (6,4) for EC (4 data stripes and 2 parity stripes) and 3-way replication for REP.
6.4. Evaluation

Table 6.3: Trace characteristics.

<table>
<thead>
<tr>
<th>Parameters</th>
<th>ycsb-zipf</th>
<th>mds-0</th>
<th>web-1</th>
<th>usr-0</th>
<th>hm-0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reqs. cnt</td>
<td>1.2M</td>
<td>1.3M</td>
<td>1.3M</td>
<td>2.2M</td>
<td>4.0M</td>
</tr>
<tr>
<td>Dataset(GB)</td>
<td>10.4</td>
<td>3.1</td>
<td>3.8</td>
<td>2.5</td>
<td>1.9</td>
</tr>
<tr>
<td>Reqs. Data(GB)</td>
<td>55</td>
<td>44</td>
<td>18</td>
<td>194</td>
<td>135</td>
</tr>
<tr>
<td>Write ratio</td>
<td>81.1%</td>
<td>93.2%</td>
<td>76.9%</td>
<td>83.6%</td>
<td>86.6%</td>
</tr>
</tbody>
</table>

Table 6.4: Test schemes.

<table>
<thead>
<tr>
<th>Schemes</th>
<th>Technique details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chameleon</td>
<td>Implement two wear balancing techniques: ARPT and HCDS</td>
</tr>
<tr>
<td>EDM</td>
<td>Implement a data migration based wear balancing technique</td>
</tr>
<tr>
<td>REP-baseline</td>
<td>Apply only REP without any wear balancing technique</td>
</tr>
<tr>
<td>EC-baseline</td>
<td>Apply only EC without any wear balancing technique</td>
</tr>
<tr>
<td>REP+EC-baseline</td>
<td>Apply Hybrid REP/EC without any wear balancing technique</td>
</tr>
</tbody>
</table>

6.4.2 Experimental Methodology

Traces. We use two kinds of workloads for our tests: YCSB workload with Zipf-like access pattern [81] and four block-level traces from MSR-Cambridge data center servers [139]: YCSB.zipf, mds_0, web_1, usr_0, and hm_0. YCSB trace is generated by YCSB benchmark which is often used to evaluate the performance of different key-value stores and cloud serving stores [81]; MSR traces are collected at the block device level from Microsoft Cambridge. Table 6.3 shows the details of trace characteristics, such as, total request count (Reqs. cnt), total dataset size (Dataset (GB)), total R/W request data (Reqs. Data(GB)), and write ratio.

Evaluated Schemes. We evaluated Chameleon by comparing it with multiple different schemes as shown in Table 6.4. To compare the state-of-art redundancy techniques, we implemented a hybrid REP/EC baseline scheme named REP+EC-baseline without using any wear balancing technique, similar to HDFS-RAID [58]. REP+EC-baseline replicates recently created data, and converts cold date from REP to EC. We also tested other two baselines schemes that applies only REP and only EC separately without using any wear leveling, denoted as REP-baseline and EC-baseline.

To compare the state-of-art wear balancing technique, we implemented and evaluated a data migration based wear balancing technique called EDM [144](detailed in ??). Note that EDM does not consider the impact of redundancy schemes on wear balancing, so we only applied a single redundancy scheme to EDM scheme, either REP or EC.
Chapter 6. Adaptive Wear Balancer for Flash Clusters

Figure 6.4: Wear variance.

**Wear Balance** To evaluate the wear variance of flash cluster, we calculate the standard deviation of the total erasure counts along with the average erasure count across 50 flash servers. Figure 6.4 shows the results of using three baseline redundancy schemes and two wear balancing schemes. The Y-axis shows the average erasure counts across 50 flash servers. Error bars represent one standard deviation.

First consider the results of three baseline schemes without using wear balancing algorithm as shown in Figure 6.4(a). Among these three baseline schemes, EC-baseline’s standard deviation error bars were much smaller than that of the two baseline schemes. This is mainly because EC naturally reduces the storage overhead by eliminating redundant copies and EC can distribute data more evenly than replication since we use RS(6, 4) for EC while 3-way replication for REP. REP+EC-baseline’s standard deviation errors were almost similar to that of REP-baseline. This is because REP+EC-baseline replicates all the newly created data and converts replicas to stripes only after they are cool down.

To compare Chameleon with EDM in terms of balancing the erasure count across 50 servers, we applied EC for the request data and evaluated EDM scheme and Chameleon scheme respectively. The reason we chose EC is that EC can achieve a smaller wear variance than both REP and REP+EC-baseline as shown in Figure 6.4(a). As shown in Figure 6.4(b), although EDM did improve the deviation of erasure counts, Chameleon significantly outperformed EDM under all workloads. For example, the standard deviation for the Chameleon scheme was at most \( \sim 1,000 \) under workload \( Hm_0 \) while the standard deviations were 1,880 and 2,316 for EDM and EC-baseline respectively as shown in Figure 6.4(b). For the two workloads, \( Web_1 \) and \( YCSB_{zipf} \), which exhibit relatively smaller standard deviation error bars compared to others, Chameleon also delivered a better wear balance compared with EDM. In particular, its standard deviations were 162 and 704 under workloads \( Web_1 \) and \( YCSB_{zipf} \) respectively while that of the EDM were 190 and 876, respectively.

Overall, Chameleon can reduce wear variance by 52% on average and at-most 81%, compared to EC-baseline. Chameleon can reduce the wear variance by 43% on average and at-most 70%, compared to EDM.
6.4. Evaluation

Flash endurance. To evaluate flash endurance, we calculate the aggregate erase cycles for all flash servers. The results are shown in Figure 6.5. The Y-axis shows cluster-wise total erasure counts. As shown, the total erase cycles when replaying the workload web_1 is relatively lower than that when replaying others. This is because that workload web_1 has lower amount of write request data than other workloads as shown in Table 6.3.

As shown in Figure 6.5(a), we observe that among three redundancy policies, REP (shown as REP-baseline) has more erasure count than other two redundancy policies because REP writes almost $3 \times$ more data to the whole cluster and entails more erasure count. While EC-baseline has the lowest erasure count since it consumes less storage than REP. The total erasure count of REP-baseline is $\sim 2 \times$ higher than that of EC-baseline.

To compare Chameleon with EDM about their compact on flash endurance across 50 servers, we applied EC for the request data and evaluated EDM scheme and Chameleon scheme, respectively, since EC can achieve a smaller total erasure count than other two redundancy policies, REP and REP+EC-baseline. Comparing Chameleon with EC-baseline scheme, we observe that Chameleon has a similar amount of cluster-wise aggregate block erase cycles with EC-baseline scheme while EDM has a significant higher total erasure count than both Chameleon and EC-baseline as shown in Figure 6.5(b). This is because Chameleon introduces less writes to the destination servers compared with EDM by using late EC/REP and EWO techniques. For EDM, the data migration process introduces significant wear overhead to the flash cluster due to extra write overhead. As we can see in Figure 6.5(b), the block erasure count of EDM is increased by up to $\sim 20\%$ under workload Usr_0, Mds_0, and Hm_0 due to data migration.

Impact on SSD write latency. We measured the average write response time in each SSD simulator to see the impact of wear balancing on write performance as GC has a significant influence on write performance as shown in Figure 6.6. Note that the write latency is measured as the time interval between SSD simulator receiving a write request and finishing the write request. Y-axis shows normalized write latency.
The write latency normalized to REP-baseline are shown in Figure 6.6(a). As shown, the average write response time when replaying the workload Web_1 is relatively lower than that when replaying others. This is because that workload Web_1 have lower amount of write request data than others as shown in Table 6.3. Moreover, EC-Baseline’s average write response time is the highest among three redundancy schemes. The write latency of EC-baseline is 1.12 on average and at-most 1.35 higher than that of REP-baseline. This is because, under EC, the writes are scattered across multiple servers (e.g., 6 in RS-(6,4)) at a smaller stripe granularity, while REP performs writes at a bigger object-level and therefore has a higher sequentiality of writes. With increasing sequentiality of writes (Figure 6.6(a)), the write performance of SSDs is observed to be improved.

To compare Chameleon with EDM with respect to impact on write performance, we applied REP for the request data and evaluated EDM scheme and Chameleon scheme since REP can achieve a better write performance than both EC and REP+EC-baseline. Compared to EDM, Chameleon has a better write performance. Chameleon can reduce the write latency by 25% on average and at-most 33%, compared to REP-baseline. In contrast, EDM can only reduce the write latency by 7% on average and at-most 20%, compared to REP-baseline. This is because Chameleon can achieve a good wear balance with minimum extra overhead. In contrast, EDM introduces considerable extra overhead during wear balancing process.

**Impact on write amplification.** We measured the write amplification (WA) after GC starts in each SSD simulator to see the impact of wear balancing on write amplification. The results are shown in Figure 6.7. Y-axis shows the write amplification.

EC-Baseline’s WA is the highest among three redundancy schemes as shown in Figure 6.7(a). We compare the WAs of EC-baseline and REP-baseline. The WA of EC-baseline is 2.11 on average and at-most 2.8, while that of REP-baseline is 1.4 on average and 1.7 at-most. The reason is the same as that of write latency: under REP, the writes have higher sequentiality because REP performs writes at a bigger object-level while the writes are scattered across multiple servers (e.g., 6 in RS-(6,4)) at a smaller stripe granularity under EC. With increasing
### 6.4. Evaluation

Sequentiality of writes (Figure 6.7(a)), the WA of SSDs improved.

Theoretically, write amplification can be defined as $1/(1-\mu)$, where $\mu$ is the utilization of victim block that needs to be cleaned during GC. That is, to make room for $(1-\mu)$ new writes, $\mu$ valid pages need to be rewritten so the total number of writes is $(1-\mu)+\mu = 1$. Thus, write amplification is directly affected by the victim block utilization. Redundancy policy impacts victim blocks’ utilization by changing the size and destination of write requests. Moreover, the relationship between redundancy policy and write amplification is not a simple linear relationship as shown in Figure 6.7(a).

To compare Chameleon with EDM about the compact on WA, we applied REP for the request data and evaluated EDM scheme and Chameleon scheme since REP can achieve a lower WA than both EC and REP+EC-baseline. Compared with EDM, Chameleon has a lower WA. Chameleon can reduce the WA by 12% on average and at-most 20%, compared to REP-baseline. While EDM can only reduce the WA by 6% on average and at-most 13%, compared to REP-baseline. There are several reasons for this behavior. First, Chameleon achieves a better wear balance distribution, which mitigates the overall write amplification due to garbage collection. For one hand, the utilization of hot flash servers is reduced. For the other hand, Chameleon introduces less writes to the destination servers compared with EDM.

#### Data state changes over time.

As discussed in Section 6.3, Chameleon achieves better wear balance by using two adaptive wear balancing techniques, ARPT and HCDS. ARPT converts data redundancy polices while HCDS off-loads data via EWO. Consequently, data has two redundancy states, REP and EC, and four intermediate states, Late REP, Late EC, REP-EWO, and EC-EWO.

To see how the data state changes over time, we calculated the aggregate amount of data in different states individually for each hour. Figure 6.8 shows data state changes over 85 hours by replaying workload $YCSB−zipf$. Y-axis shows the percentage of data in different states. As shown, We combine the REP-EWO and EC-EWO together as EWO state since
Figure 6.8: Data state changes over 85 hours under Chameleon by replaying YCSB-zipf workload.

The amount of data in REP-EWO state is roughly similar to that of data in EC-EWO.

First, all the data started with EC state since we applied EC for all newly created data and after three hours, ARPT started to convert a small mount of hot data from EC to late REP and later cover to REP when their update requests come. During the 5th hour, Chameleon detected that wear imbalance happened and started HCDS to swap hot data with cold data for wear balancing. After that we see the data in EWO state increased up to 20% during the 20th hour and then fluctuated during the period of the 25th-65th hour. During this period, up to 20% of data was involved in HCDS for wear balancing. The slight decrease during this period means that a certain amount of data was offloaded and covered to a redundancy state.

After the 65th hour, we see a decrease for the amount of data in EWO state, implying that not only the wear but also the workload were almost balanced. Moreover, the data involved in HCDS was almost converted from an intermediate EWO state to a final redundancy state.

Overall, we can see the data involved in HCDS was less than 20% for each hour, while the data involved in ARPT was less than 5% for each hour. We conclude that only a relative small amount of data’s popularity changes and HCDS plays a major role in wear balancing.

6.5 Chapter summary

We have presented the design and implementation of Chameleon, a wear balancer for distributed flash-based storage cluster. Chameleon aims to improve both the flash endurance and runtime performance. First, Chameleon takes advantages of the out-of-place update feature of flash memory by directly offloading the writes/updates across flash servers instead of moving data across flash servers to mitigate extra-wear cost. We implemented several optimizations to this end: late replicating (Late REP), late encoding (Late EC), and endurance aware write offloading (EWO). Second, Chameleon provides two adaptive wear balancing algorithms, namely, redundancy policy transition (ARPT) and Hot/Cold data swapping
(HCDS) to balance the wear distribution across the flash servers, coupled with write offloading and redundancy policies to balance the erasure count and improve both lifetime and performance. Evaluation shows that Chameleon reduces the wear distribution deviation by up to 81%, while improving the write performance by up to 33%. In the future, we aim to realize Chameleon in real flash hardware such as Open-channel SSD [21] and integrate Chameleon to other distributed storage types such as distributed file systems.
Chapter 7

Conclusion and Future Work

A key goal of this dissertation is to make innovative storage solutions for containerized applications. In this dissertation, we first provided a comprehensive and large-scale characterization of the images and layers downloaded from Docker Hub [7]. Moreover, we also designed an adaptive wear balancer for SSD-based storage clusters to improve the performance and lifespan of the flash cluster that can be used for container clusters.

Based on the image characterization, we investigated the performance implications for optimizing the design of Docker container storage driver and storage interface driver, respectively. Specifically, we perform different characterizations of the performance impact among a wide range of container storage drivers as well as different container storage interface drivers on different storage clusters (i.e. HDD- and SSD-based storage clusters). Furthermore, we designed a flexible high-performance deduplication for Docker registries to meet different storage space savings and layer pulling performance requirements.

This dissertation is driven by the relentless growth of container image dataset and the challenges to storage, the growing demand of high-performance deduplication for Docker registries, and tailoring storage systems for different containerized applications. This dissertation proposes, designs, and implements a series of novel techniques, algorithms, and frameworks that can improve the performance. This dissertation adopts three steps: (1) analyzing the Docker image dataset; (2) deriving the design implications; (3) designing a new storage framework. The overarching goal of this dissertation is to improve the flexibility, performance, and space-efficacy of storage systems for modern containerized applications by using data analysis, deduplication, and wear balancing.

The experimental evaluation demonstrates the effectiveness of our optimizations and techniques to make storage systems flexible and space-efficacy.
7.1 Summary

Containers images are stored and distributed via a Docker registry. As the container market continues to expand, Docker registries have to manage a growing number of images and layers. As a result, a comprehensive and large-scale characterization of the images and layers stored in the Docker registry is needed to find interesting properties regarding storage, layer sharing, and redundancy. These properties have implications on the future design of Docker registry and container storage drivers. Moreover, storage is critical for container-based applications and container storage drivers largely affect container runtime performance, especially I/O performance. Therefore, there is need to perform performance analysis on storage driver and optimize the storage design for containerized applications. This dissertation along this line tackles the above problems using dataset analysis, workloads analysis, and performance analysis that cohesively guide storage optimizations with the goal of maximizing both performance, efficiency, and space-efficacy.

We first carried out the first comprehensive analysis of container images stored in Docker Hub. We analyzed a compressed 47 TB dataset resulting in over 1 million layers and 5 billion files. We studied a variety of storage metrics on layers, images, and files. Our findings reveal that there is room for optimizing how images are stored and used. For example, file-level deduplication can be used to improve space efficacy and caching popular layers can reduce disk I/O.

Second, based on our image characterizations, we further investigated the I/O performance of storage drivers. Starting with layer pulling/pushing analysis, we found that compression methods and storage options for unpacked layer contents can both become bottlenecks. Various optimizations are proposed to speed up layer pulling/pushing. Moreover, we drew several design implications by analyzing container storage drivers using a wide range of block sizes.

Third, we exploit the high file redundancy in real-world container images to drastically reduce the demanding storage requirements of the growing registries. We design Duphunter, a new Docker registry architecture that provides flexible and high performance deduplication for container images and reduces storage utilization. Duphunter supports multiple configurable deduplication modes to meet different space saving and performance requirements. Additionally, it parallelizes layer reconstruction locally and across the cluster to further mitigate overheads. Moreover, by exploiting knowledge of the application domain, Duphunter introduces a two-tier storage hierarchy with a novel layer prefetch/pre-construct cache algorithm based on user access patterns.

Finally, we design and implement Chameleon, a wear balancer for distributed flash-based storage cluster. Chameleon aims to improve both the flash endurance and runtime performance. First, Chameleon takes advantages of the out-of-place update feature of flash memory by directly offloading the writes/updates across flash servers instead of moving data across flash servers to mitigate extra-wear cost. We implemented several optimizations to this end:
late replicating (Late REP), late encoding (Late EC), and endurance aware write offloading (EWO). Second, Chameleon provides two adaptive wear balancing algorithms, namely, redundancy policy transition (ARPT) and Hot/Cold data swapping (HCDS) to balance the wear distribution across the flash servers, coupled with write offloading and redundancy policies to balance the erasure count and improve both lifetime and performance.

7.2 Future Directions

This dissertation are focused on problems that exists in container storage systems. We are particularly in designing storage systems with high performance and flexibility and better support large-scale container clusters, and extend our understanding in serverless, data analytics, and distributed machine learning. We discuss several future directions as an extension to this dissertation.

7.2.1 Low-overhead container network

The adoption of microservices architecture in application development and deployment is growing rapidly due to its increased agility and scalability. Containers are used in microservices architecture to revolutionize application development and delivery. With microservices, an application is refactored into separate services that are loosely coupled container instances and communicate via RPCs. In container-based applications, although services are modeled as isolated units, services often depend on other services for remote data or computation via RPCs. Moreover, compare to host network, container overlay networks impose significant overhead due to extra network stack traverse and packet encapsulation/decapsulation [201]. [94] conducted a performance analysis on different microservices workloads and observed 5-75% of execution time spent on networking. Consequently, networking overhead accounts a large portion of application execution time in container clusters [94, 201].

7.2.2 High-performance storage system for serverless analytics

Currently, serverless computing becomes a new paradigm for Cloud computing. Many cloud providers have proposed services that execute functions in the cloud (i.e., Function-as-a-Service). These functions are subject to duration limit and resource limit, which are suitable for computations like analytics workloads [155] for high utilization. However, the communications between worker functions, mapping tasks to different storage medias, and handling intermediate data present big challenges to serverless framework design. We will optimize the storage system for serverless analytics.
7.2. Future Directions

7.2.3 Tailoring serverless computing for distributed machine learning

Serverless computing has recently emerged as a compelling computing paradigm to address the resource utilization problem of data center [70, 86, 174, 180, 185, 190]. Serverless computing relies on stateless functions submitted by users and automatically schedule them to different compute nodes with different resources. In this case, serverless computing is a promising approach to address the problem of resource provision for machine learning users. However, existing serverless frameworks have several limitations, for example, limited amount of memory and storage, low network bandwidth, etc. We will tailor the serverless computing for distributed machine learning.
Bibliography

[22] Use the overlayfs storage driver-how the overlay2 driver works. https://docs.docker.com/storage/storagedriver/overlayfs-driver/#how-the-overlay2-driver-works.


[68] Ceph. [https://docs.ceph.com/docs/master/dev/deduplication/](https://docs.ceph.com/docs/master/dev/deduplication/).


[71] Li-Pin Chang. On efficient wear leveling for large-scale flash-memory storage systems. In *ACM SAC ’07*. 
[72] Li-Pin Chang and Tei-Wei Kuo. An efficient management scheme for large-scale flash-memory storage systems. In *ACM SAC ’04*.


[83] Datadog. 8 Surprising Facts about Real Docker Adoption. [https://www.datadoghq.com/docker-adoptions/](https://www.datadoghq.com/docker-adoptions/).


[85] Biplob Debnath, Sudipta Sengupta, and Jin Li. Flashstore: high throughput persistent key-value store. *VLDB ’10*.


BIBLIOGRAPHY


[142] OpenStack Swift storage driver. Openstack swift storage driver. https://docs.docker.com/registry/storage-drivers/swift/.


