Stony Brook University

Efficient Implementation Techniques for Block-Level Cloud Storage Systems

A Dissertation Presented

by

Dilip Nijagal Simha

to

The Graduate School

in Partial Fulfillment of the Requirements

for the Degree of

Doctor of Philosophy

in

Computer Science

Stony Brook University

May 2014

Stony Brook University

The Graduate School

Dilip Nijagal Simha

We, the dissertation committee for the above candidate for the Doctor of Philosophy degree, hereby recommend acceptance of this dissertation.

Dr. Tzi-cker Chiueh – Dissertation Advisor
Professor, Department of Computer Science

Dr. Erez Zadok – Chairperson of Defense
Associate Professor, Department of Computer Science

Dr. Donald E. Porter
Assistant Professor, Department of Computer Science

Dr. Marcos K. Aguilera
Senior Researcher, Microsoft Research, Silicon Valley

This dissertation is accepted by the Graduate School.

Charles Taber
Dean of the Graduate School

Abstract of the Dissertation

Efficient Implementation Techniques for Block-Level Cloud Storage Systems

by

Dilip Nijagal Simha

Doctor of Philosophy

in

Computer Science

Stony Brook University

2014

A fundamental building block for an IaaS (Infrastructure-as-a-Service) cloud service such as Amazon's EC2 is a storage virtualization system that provides block-level storage services to individual virtual machines over the network. This dissertation addresses four major problems in such a block-level cloud storage system, in the context of an end-to-end IaaS solution called ITRI Cloud OS. First, to effectively eliminate redundancies in stored data blocks, we propose a scalable block-level deduplication engine called *Sungem*, which uses both sampling and prefetching to minimize the performance overhead of fingerprint accesses, and features a storage block garbage collection algorithm whose runtime overhead is proportional only to the size of the delta between consecutive backup operations. Second, to efficiently flush metadata updates associated with large-scale block-level storage management, we developed a novel storage system architecture called *BOSC* (Batching mOdifications with Sequential Commit), which uses largely sequential writes to commit updates to disk and is

thus able to sustain high-throughput and low-latency metadata updates that are largely random. Third, as part of the BOSC architecture, we invented a high-throughput low-latency disk logging system called *Beluga*, which fashions a carefully tuned disk write pipeline and makes it possible to provide, on an array of three commodity 7200 RPM SATA disks, close to 5 million fine-grained (64-byte) disk logging operations per second, which is close to the maximum possible bandwidth on a commodity disk, while keeping the latency of each logging operation under 1 msec. Finally, we devised a set of techniques for supporting software-defined storage service on a distributed and replicated storage architecture. Specifically, we developed a distributed storage QoS guarantee system called *Cheetah*, which is able to provide a bandwidth guarantee to each virtual disk attached to a virtual machine, while ensuring the loads on the distributed storage nodes be balanced, and the locality of the access stream associated with each virtual disk be preserved as much as possible.

Dedicated to:

Dad, Mom, Kruthi,
Smitha, Anand, Samarth and Diya.

Contents

List of Figures xi

List of Tables xiv

Acknowledgements xvi

1 Introduction 1
- 1.1 Cloud Storage . . . 1
  - 1.1.1 Generic Challenges in Cloud Storage System . . . 3
  - 1.1.2 Challenges in Block-Level Cloud Storage System . . . 4
  - 1.1.3 Challenges in Cloud Storage's Back-End Management . 5
- 1.2 SDDS System Architecture . . . 7
  - 1.2.1 Block Address Namespace . . . 8
  - 1.2.2 Data Path Management . . . 10
  - 1.2.3 Comprehensive Data Protection . . . 10
  - 1.2.4 Dirty Block Tracker . . . 12
- 1.3 Challenges Addressed by this Dissertation . . . 12
  - 1.3.1 What this Dissertation is Not About . . . 15
- 1.4 Research Contributions . . . 16
- 1.5 Dissertation Outline . . . 18

2 Related Work 20
- 2.1 Cloud Storage . . . 20
- 2.2 Deduplication Techniques . . . 24
  - 2.2.1 Content-Addressable Storage . . . 24
  - 2.2.2 Timing of Backup . . . 25
  - 2.2.3 Data Comparison Techniques . . . 26
  - 2.2.4 Granularity of Deduplication . . . 26
  - 2.2.5 Positioning of Deduplication . . . 28
  - 2.2.6 Variable Segment vs Fixed Segment . . . 30
  - 2.2.7 Faster Index Lookup Strategies . . . 32

2.2.8 Medium of Backup Storage . . . 35
2.2.9 Deduplication Trace Workload Analysis . . . 36
2.3 Garbage Collection Techniques for Deduplication Storage Systems 37
2.3.1 Mark and Sweep . . . 37
2.3.2 Reference Count based . . . 38
2.3.3 Expiry Time based . . . 39
2.3.4 Summary of GC comparisons . . . 40
2.4 Fast Random Updates to On-Disk Data Structures . . . 40
2.5 Fast Disk Logging . . . 42
2.6 QoS for Distributed Storage Systems . . . 45
2.6.1 Description of QoS specification in Service Level Agreements . . . 46
2.6.2 Granularity of QoS Enforcements . . . 46
2.6.3 Location of Collecting Statistics . . . 47
2.6.4 Performance Isolation . . . 49
2.6.5 Provisioning Hardware Resources . . . 50
2.6.6 Load Balancing . . . 50
2.6.7 Extending QoS Ideas From Non-Storage Systems . . . 52

3 Scalable Deduplication and Garbage Collection 54
3.1 Introduction . . . 54
3.2 Our Approach . . . 57
3.2.1 System Architecture . . . 57
3.2.2 Fingerprint Segmentation and Placement . . . 58
3.2.3 Variable Fingerprint Sampling . . . 59
3.3 Scalable Garbage Collection . . . 60
3.3.1 Hybrid GC: Our Approach . . . 61
3.3.2 Batched Updates to P-Array . . . 62
3.4 Parallelization Techniques for Deduplication and Garbage Collection . . . 63
3.4.1 Distributed Deduplication Algorithm Design . . . 64
3.4.2 Distributed GC Design . . . 65
3.5 Performance Evaluation . . . 67
3.5.1 Evaluation Methodology . . . 67
3.5.2 Overall Performance . . . 69
3.5.3 Effectiveness of Sampled Fingerprint Index . . . 72
3.5.4 Content Proximity-Based Fingerprint Placement . . . 73
3.5.5 Garbage Collection Overhead . . . 75
3.5.6 Effectiveness of Container Cache . . . 76
3.5.7 Impact of Controlling Stored Segment Formation . . . 76
3.5.8 Parallel Deduplication tradeoffs . . . 77

3.6 Summary . . . 78

4 A Trace-based Study for Deduplication Algorithm Design 80
4.1 Trace Collection and Conversion . . . 81
4.1.1 Trace Collection . . . 81
4.1.2 Trace Conversion . . . 82
4.1.3 Trace Processing . . . 83
4.2 General Duplicity Pattern . . . 85
4.3 Trace-based Deduplication Design Tradeoff Analysis . . . 88
4.3.1 Sampling of Stored Segments . . . 89
4.3.2 Placement of Stored Fingerprint Segments . . . 89
4.3.3 Garbage Collection of Fingerprints . . . 93
4.4 Impact of Deduplication Granularity . . . 94
4.5 To BF or Not to BF . . . 96
4.6 Summary . . . 97

5 An Update-Aware Storage System for Low-Locality Update-Intensive Workloads 99
5.1 Update-Aware Disk Access Interface . . . 102
5.1.1 Caveats with Call-back Function . . . 103
5.2 BOSC Architecture . . . 104
5.2.1 Low-Latency Disk Logging . . . 104
5.2.2 Sequential Commit of Aggregated Updates . . . 105
5.2.3 Recovery Processing . . . 107
5.2.4 Extensions . . . 108
5.3 Applications of BOSC . . . 111
5.3.1 BOSC-Based B^+ Tree . . . 111
5.3.2 Hash Table . . . 113
5.4 Performance Evaluation . . . 113
5.4.1 Evaluation Methodology . . . 113
5.4.2 Logging disk, Data disk combinations . . . 114
5.4.3 Overall Performance Improvement on B^+ Tree . . . 116
5.4.4 Overall performance improvement on Hash Table . . . 123
5.4.5 Application of BOSC to *Mariner* . . . 125
5.5 Summary . . . 126

6 High Throughput Low Latency Disk Logging 128
6.1 Vanilla Disk Logging . . . 130
6.2 Toy-Train Disk Logging . . . 132
6.2.1 Conceptual Model . . . 132
6.2.2 Application Programming Interface . . . 133

6.2.3 Streamlined Disk Write Pipeline . . . 134
6.2.4 Dense-Mode Logging . . . 137
6.2.5 Sparse-Mode Logging . . . 140
6.3 Performance Evaluation . . . 143
6.3.1 Methodology . . . 143
6.3.2 Dense-Mode Logging . . . 144
6.3.3 Sparse-Mode Logging . . . 151
6.3.4 Comparison with SSD-based Logging . . . 153
6.4 Summary . . . 154

7 Quality of Service Guarantee for Software-Defined Distributed Storage Systems 155
7.1 SDDS System Architecture in the Context of Managing QoS Functionality . . . 155
7.1.1 System Model . . . 157
7.1.2 Service Model . . . 157
7.2 *Cheetah*'s Objectives, Challenges & Solution . . . 158
7.2.1 Design Objectives . . . 158
7.2.2 Technical Challenges . . . 158
7.2.3 Solution Overview . . . 163
7.3 Quantification of Physical Disk Resource Requirements . . . 164
7.4 Read Load Balancing . . . 167
7.4.1 RLB Algorithm . . . 169
7.4.2 RLB Integration with *Cheetah* . . . 171
7.5 Flow Control . . . 173
7.6 CFVC: A QoS Aware Disk Scheduler . . . 176
7.6.1 CFVC Scheduler Algorithm . . . 176
7.6.2 Implementation Challenges Integrating CFVC Scheduler into *Cheetah* . . . 180
7.7 Putting it All Together . . . 184
7.8 Evaluation Methodology . . . 185
7.8.1 Current Prototype . . . 185
7.8.2 DA Simulator for RLB Evaluation . . . 188
7.8.3 Synthetic Trace Generation . . . 195
7.9 Performance Evaluation of the Automated PB Extraction Process 196
7.9.1 Effect of Workload Locality on PB using Real-World I/O Trace . . . 196
7.9.2 Effect of Workload Locality on PB using Synthetic Workload . . . 198
7.9.3 Effectiveness of PB Extraction Process on a Shared DA 200
7.10 Evaluation of the Effectiveness of Bandwidth Decomposition . 202

7.10.1 Variable VD-DA Mappings 202
7.10.2 Variations in Read/Write Ratio 205
7.10.3 Variations in Sequential Locality 205
7.10.4 Short Term Variations in Workload Locality 208
7.10.5 Centralized RLB Scheduler's Processing Time 209
7.11 Performance Evaluation of Per-VD Scheduler 209
7.12 Summary 212

8 Conclusion and Future Directions **214**
8.1 Conclusion 214
8.2 Future Directions 216
8.3 Final Words 219

Bibliography **220**

List of Figures

1.1 SDDS Architecture 7
1.2 Addressing namespace in DISCO 9

2.1 Sampling and Prefetching in dedupe 32

3.1 *Sungem* Architecture 57
3.2 Metadata Updates in Garbage Collection 62
3.3 *Sungem*'s deduplication throughput and ratio 70
3.4 Reference count distribution in *Sungem*'s trace 71
3.5 *Sungem*'s SFI effectiveness 72
3.6 *Sungem*'s container cache effectiveness 77
3.7 Distributed *Sungem* performance 78

4.1 Dedupe trace conversion scheme 82
4.2 *Sungem*'s TP vs CP visual comparison 91
4.3 *Histogram showing the temporal distance between the matched blocks* 93
4.4 Retention Period vs Duplicity in detailed trace 93

5.1 BOSC Architecture 104
5.2 BOSC2 extension with swapping disks 109
5.3 Insertion and read throughput for various logging disk/data disk variations 114
5.4 BOSC vs vanilla comparison for random insert workload in B^+ tree 116
5.5 BOSC vs vanilla comparison for random update workload in B^+ tree 117
5.6 Effectiveness of BOSC-based B^+ tree in-memory queue and low latency logging 119
5.7 BOSC-based B^+ tree sensitivity study with variations in leaf node size 120
5.8 BOSC-based B^+ tree sensitivity study with variations in record size 121

5.9 BOSC-based B^+ tree sensitivity study with variations in both leaf node size and record size . . . 122
5.10 BOSC-based B^+ tree sensitivity study with variations in index size . . . 123
5.11 BOSC-based B^+ tree sensitivity study with variations in in-memory queue size . . . 124
5.12 BOSC vs vanilla comparison for random insert workload in hash table . . . 125
5.13 BOSC vs vanilla comparison for random update workload in hash table . . . 126
5.14 Disk write logging performance of Mariner with and without BOSC comparison . . . 127

6.1 *Beluga* Architecture . . . 136
6.2 Sparse mode *Beluga* logging example . . . 142
6.3 Adaptive Batch Size Selection in *Beluga* . . . 149
6.4 *Beluga* sensitivity study: Variations in NCQ length . . . 150

7.1 *Detailed overview of the components of a VDC* . . . 156
7.2 *Illustration of DRUT computation on a DA shared by multiple VDs* 164
7.3 *Illustration of a scenario where a naive greedy RLB algorithm fails to load balance the DAs* . . . 167
7.4 Illustration of RLB management . . . 170
7.5 Illustration of Flow Control management . . . 175
7.6 Illustration of short term unfairness problem in a typical VC disk scheduler algorithm . . . 180
7.7 *Simulation setup for evaluating the effectiveness of PB extraction* . 186
7.8 Charts with load distribution for uneven VD-DA mappings . . 202
7.9 Figure comparing RLB, RR and RND schedulers for random mappings between 100VDs and 40 DAs . . . 204
7.10 Figure comparing RLB, RR and RND schedulers for random mappings between 200VDs and 70 DAs . . . 204
7.11 Figure comparing RLB, RR and RND schedulers for random mappings between 600VDs and 200 DAs . . . 204
7.12 Figure comparing RLB, RR and RND schedulers for random mappings between 800VDs and 250 DAs . . . 204
7.13 Comparsion of RR, RND and RLB schedulers for read/write variations . . . 206
7.14 Charts with load distribution for variations in sequential locality in input workload . . . 207

7.15 Charts with load distribution for short term variations in input workload 208
7.16 Charts comparing locality unaware vs locality aware RLB for a low locality workload 210
7.17 Charts comparing locality unaware vs locality aware RLB for a high locality workload 210

List of Tables

2.1 GC comparison analysis 37
2.2 White-box vs Black-box stats collection techniques 47

3.1 *Sungem*'s real-world trace composition 68
3.2 *Sungem*'s TP vs CP comparison 74
3.3 *Sungem*'s GC performance 75
3.4 *Sungem*'s GC detailed performance 76
3.5 *Sungem*'s K-factor effectiveness 77

4.1 File-grouping classification in detailed real-world dedupe trace 85
4.2 Block distribution in detailed dedupe trace 86
4.3 Duplicity of blocks internal and external to a file 87
4.4 Duplicity vs File Size analysis 88
4.5 *Size distribution of recurring stored fingerprint segments* 90
4.6 *Sungem*'s TP vs CP comparison on the detailed trace 90
4.7 Impact of deduplication granularity on duplicity 94
4.8 Impact of deduplication granularity on duplicity for each file type 95
4.9 Performance impact of RBF in *Sungem* 97

5.1 BOSC-based B^+ tree sensitivity study with read query latency 122

6.1 Latency and throughput of file-based and device-based (Raw) disk logging 131
6.2 *Beluga* performance with variations in batch size 145
6.3 *Detailed breakdown of the time each logging operation spends in the disk write pipeline as the batch size is varied* 147
6.4 Effect of record size on *Beluga*'s performance 147
6.5 *Beluga*'s performance at various offsets on disk 148
6.6 *Beluga* with multiple disks 149
6.7 Performance of sparse mode *Beluga* 151
6.8 Detailed performance evaluation of sparse mode *Beluga* 153
6.9 Performance of SSD logging 153

7.1 Table showing the correctness of simulated DA 194
7.2 Table showing the correctness of simulated DA using negative result 194
7.3 Variations in PB with workload locality in a real-world trace . 197
7.4 Variations in PB with workload locality in a real-world trace, when request generation rate is doubled 197
7.5 Variations in physical bandwidth with request generation rate using synthetic workload 198
7.6 Variations in physical bandwidth with workload locality in a synthetic trace 199
7.7 Variations in physical bandwidth with read/write ratio in a synthetic trace 200
7.8 Table showing the effectiveness of PB extraction process on a shared DA 200
7.9 Table showing the effectiveness of PB extraction process on a shared overloaded DA 201
7.10 Table showing the time taken by centralized RLB scheduler for varying number of VD-DA configurations 209

Acknowledgements

Thank you, God, for instilling the belief in me to take up this challenging journey, tolerance to endure it, and rewarding me in time.

I would like to extend my sincere gratitude to my advisor, professor Tzicker Chiueh, who has been extremely supportive throughout my five years of graduate studies, on academic as well as non-academic matters. It is my privilege to be technically trained by him because his discipline and commitment towards top notch System's research is unparalleled. The amount of flexibility, trust and challenges he entrusts in a graduate student is commendable. Despite his busy schedule, he has always been there for me, guiding me, involving me in intellectually stimulating discussions and incessantly pushing me to take that extra leap towards reaching my goals. There has never been an occasion when a discussion with him has not yielded a positive result. He has always been and will continue to be my role model. I wish to thank him for aligning my career in a path filled with challenging opportunities, by continuing to contribute aggressively towards System's research.

I thank Dr. Don Porter for advising me on several issues including on structuring presentations, tips on writing techniques and improving my dissertation. He has always been extremely kind, cooperative and approachable. I thank my other committee members Dr. Erez Zadok and Dr. Marcos Aguilera for agreeing to be part of the dissertation committee.

I thank all my colleagues and superiors at the CCMA division of ITRI, Taiwan, for all the professional and friendly warmth that they have extended during all my internships. My special thanks to Sandy, Christine and all the staff members for helping me focus on my research work without worrying about food and language in an environment that is completely alien to my way of living. It is because of all their love and affection that kept me visiting Taiwan for more than one internship. I also wish to thank ITRI for funding my research and conference trips.

I thank all my PhD colleagues Maohua, Steve, William and Yifeng for all the discussions related to solving research problems, finding jobs and Asian political matters! I thank my MS colleagues Ganesh, Pallav and Dileep who

contributed to various project discussions, implementations and performance evaluations.

My dad, Madhava Murthy and mom, Geetha Murthy, deserve a special thanks for always being there for me. It is not possible to express with simple words their efforts towards my progress in graduate school, for they have been the back-bone of this exciting journey. When I had to leave them alone and come to US, they never held me back. Rather, they motivated me to never give-up my ambitions until it is complete and have never left me worry for financial or emotional issues. During those times when I faced rejects after rejects to all my paper submissions, they never doubted me, but kept motivating me to push even harder. I thank you dad and mom, and I promise to be always there for you.

My brother-in-law, Anand, was instrumental in instilling in me the passion for Computer Science. If not for my sister, Smitha's constant prodding to not lose hope after my lackluster GRE score, I would have probably cut short my PhD ambitions. Thanks to Anand, Smitha and their kids Samarth and Diya for all the healthy distractions, emotional and financial support.

My wife, Kruthi, deserves a special thanks. She accepted a marriage proposal from a PhD candidate, who then didn't have a house to stay, a stable job and a salary to maintain a family, and who promised her nothing but eternal love. I am very glad that I am that PhD candidate and am short of words to describe the amount of sacrifice that she has put up. She gave up her flourishing law career in India and worked relentlessly on building her career revolving around mine. She was content with infrequent nearby outings, cost conscious shopping, and never complained about failing to keep up to my promises. Although, she didn't understand the technicalities in my projects, she was a patient listener, and has been the best critic of my presentations and writing skills. Thank you, Kruthi for this wonderful support and unconditional love.

A special thanks to my in-laws for sharing Kruthi with me, and for believing in me and my abilities to pursue my career in research. Thanks for all your prayers, blessings and encouragement.

Thanks to all my friends, relatives, well-wishers and in particular to all my 2010 batch MS graduate colleagues who have been extremely supportive and never made me miss home.

Last but not the least, I sincerely thank my first, truly inspiring guru (teacher), late Mr. Gopalakrishna, who is fondly remembered as GK sir. It was his effervescence and never give-up attitude that inspired me to dream and chase my dreams with relentless hard work. I owe you a lot and will do my bit to repay it by following your steps in sharing knowledge for the needy and the greedy.

Chapter 1

Introduction

1.1 Cloud Storage

A traditional standalone storage system uses directly attached storage (DAS) devices to offer storage related services to the applications generating I/O requests. The storage related services range from handling regular read/write I/Os to managing sophisticated quality of service (QoS) guarantees. These storage services have been studied in depth in the literature for more than a decade. With Big data [1] gaining importance in the recent years, the data storage requirements have exploded much beyond the capabilities of a traditional standalone storage system. It is evident that there isn't much a DAS device can offer in the face of massive data workloads from big data like applications because of the physical scaling limitations. In order to extend the storage capacities to large-scale, a popular approach is to move all the data to the cloud and let the cloud storage service provider manage the data. The cloud storage service provider manages a large scale data center that is optimized to support larger number of physical hardware resources. A tenant who wishes to configure the data storage management for his applications, purchases a variety of storage related services from such cloud storage service providers.

A cloud storage service consists of a front-end that provides an access interface to the tenants' applications and a back-end that manages the hardware resources that can collectively support the exposed interface. The access interface corresponds to the granularity of storage that is supported by the back-end. Based on the granularity of access to storage, cloud storage services can be categorized as block-level, object-level, file-level or database-level, and depending on the specific access interface needed by an application, tenants choose a cloud storage service accordingly. Commodity storage devices support the

block-level access granularity, and sophisticated software and hardware techniques enable higher levels of access granularity. At block-level granularity, each individual disk block can be individually accessed and different applications can configure different block sizes. With larger block sizes, the I/O throughput is maximized because of two main reasons. First, the size of metadata is typically fixed, irrespective of the block's size and hence a larger block uses lesser metadata when compared to aggregate metadata used by smaller blocks. Second, fewer network transfers are done for a larger block and that saves the precious network bandwidth. Though each network transfer for a larger block incurs higher bandwidth usage, there are various other factors like TCP/IP metadata and flow control mechanisms, that incurs additional latency for every network transfer, and such attributes can be amortized with larger block sizes. However, too large a block size will result in lesser performance of storage resources because only a fraction of the larger block can be modified during every update operation. Therefore, applications need to make an informed decision while configuring the block size. At object-level access granularity, the cloud storage service handles applications with different block-size requirements in a generic manner. Such a generic storage management supports a large variety of applications with different I/O block sizes and the object-level interface is oblivious to the type of data stored in the object. For the very same reason, the object-level interface may not give optimal performance because specific optimizations that could be possible with respect to block size or block content are no longer possible. At file-level granularity, an application need not worry about the low-level storage semantics, and instead directly stores an entire file as it is on the cloud. The cloud storage service attaches file-level semantics to every file, and extracts meaningful attributes to support advanced storage functionalities like compression and deduplication. There are various distributed file systems [2, 3] that provide an efficient file-system interface over the cloud. At an even higher granularity, an entire database is managed on the cloud. With such a service, applications move their entire storage stack to the cloud and completely outsource the data storage management to the cloud storage service provider.

The back-end of a cloud storage service manages large-scale physical storage resources and there are several ways to configure the back-end system. A popular technique is to adopt either a network attached storage (NAS) configuration or a storage area network (SAN) configuration, that projects storage devices spread over the local network as DAS devices to the back-end storage system on the cloud-scale data center. While NAS storage is offered at file-level granularity, typically over TCP/IP network, SAN storage is offered at block-level granularity, typically over fiber optic network. Both NAS and

SAN technologies use enterprise quality hardware consisting of storage devices like magnetic disks and SSDs, network switches, and a small CPU that runs a minimal operating system to control the storage devices. An alternative solution to expensive enterprise storage setup is to configure the storage disks as just a bunch of disks (JBOD) into one physical storage unit and provision a storage system with several such JBODs that effectively appear as just a bunch of JBODs, which we term as JBOJBOD. A JBOD device groups together a bunch of disks and can be configured either to concatenate all its disks to access them as a single huge volume or to access each disk individually. JBODs offer block-level access granularity. Unlike the popular redundant array of inexpensive disks (RAID) technique, JBOD device doesn't offer any redundancy, and more importantly a JBOD device works with disks of any type and size. JBOJBOD technique has picked up pace over the last few years and has been successfully deployed today in companies like Facebook, Twitter, Google, Amazon that use storage resources over gigantic scales in the range of several peta bytes.

1.1.1 Generic Challenges in Cloud Storage System

A cloud storage service handles data workload from multiple tenants and each tenant submits data workload that is generated from one or more of his applications. While a naive way to manage the cloud storage service's back-end is to allocate dedicated storage to each tenant's application, an advanced technique stores multiple tenants' data in a shared hardware environment, such that the sharing process is transparent to the applications. Though the naive solution to use dedicated hardware is relatively easier to design and manage, it is quite expensive. However, it is desirable in a few cases like when the tenant has a secure application that cannot afford security breaches of any kind. In-spite of the advancements in secure storage research [4, 5], providing high security standards in a shared data environment, it is tough to prove absolute foolproof security. There are other scenarios where-in an application has zero tolerance to noisy neighbors, which is a standard term used to describe the case where the performance of an application degrades due to large variations in data workload from another application sharing the same hardware resources as that of the affected application.

The main advantage of sharing the hardware resources among multiple tenants is the cost benefit. Higher the amount of sharing, higher is the cost-benefit. Typically, each tenant's application generates data workloads at different rates and in order to maximize the raw bandwidth of the physical storage, data workload from multiple tenants' applications can time-share the hardware resources. If the applications sharing the hardware resources generate data at

disjoint intervals in time, then the I/O performance of each application is completely independent of the data pattern in the workload of other applications. As an example, assume a tenant has two applications: web-server and a log-server. The log-server records every web-service request processed by the web-server and stores it in the cloud at periodic intervals. The web-server is more dynamic in nature and submits requests to load/store data from/to the cloud at more regular intervals. If the log-server is designed to submit data to the cloud only when the web-server is inactive, these two applications could easily share the same set of hardware resources without affecting each other's performance. If either the time-sharing is not designed efficiently, or if an application generates data workload with unpredictable pattern, then the sharing logic fails to guarantee assured performance to each application, unless specific optimizations are designed to handle such fluctuations in data workload. Therefore, it is extremely challenging to design a cloud storage service to efficiently utilize the hardware resources and simultaneously ensure high cost benefit to tenants, while guaranteeing assured I/O performance to their applications.

1.1.2 Challenges in Block-Level Cloud Storage System

Block-level access interface forms the back-bone of the cloud storage service, and interfaces with higher-level access granularities are built on top of the block-level interface using sophisticated software techniques. Therefore, it is important to ensure optimal performance of all block-level functionalities, but there are several challenges in designing a cloud-storage system at block-level granularity, and a few important ones are described below. In a large-scale cloud storage system, the number of available physical blocks range from few millions to several billions. When an application requests for storage space, an allocation agent in the storage system goes through the collection of available blocks and allocates the necessary blocks to the requesting application. The allocation policy has to be carefully designed to identify appropriate storage devices in the entire storage system for each storage allocation request, while ensuring minimal internal and external fragmentation in the storage space. Selecting the appropriate devices depends on the specific I/O requirements from the tenant's application. For example, when an application demands large disk I/O bandwidth, the allocation manager should allocate storage space, such that the available disk bandwidth on the selected storage device is sufficient to satisfy the requirements of the given application, otherwise the tenant's application will experience large unexpected latencies in the disk I/O operation, leading to dissatisfaction and possible breach in quality of service agreements with the cloud storage service provider.

Garbage collection is another important aspect in a large-scale storage system. Disk blocks are shared either explicitly or implicitly within and across the tenants' applications for various reasons. While explicit sharing is requested by the application, implicit sharing is performed by the storage system without the knowledge of the tenant, so that redundancies in data could be eliminated to save the previous storage space. The storage system also replicates the disk blocks for managing advanced quality of service features like high availability. All such data sharing patterns necessitate metadata management for each physical block and hence the garbage collection policy needs sophisticated data structures to efficiently identify and collect unused blocks in the entire storage system. For a peta-byte storage system, if each metadata is say 20 bytes long, and each physical block is of 8KB, then the metadata index is of 2.5 TB in size. Since the metadata is very important and has to be persistently stored on disk, careful design is necessary to not let the disk I/O operations on the metadata index bottleneck the garbage collection process. Additionally, the data workload on garbage collection system consists of updates to metadata index, with very low locality, and storage systems typically handle random updates poorly with very low throughput and high latency. Therefore, it is very important to handle random disk I/O operations efficiently.

The block-level storage system needs a mapping service that translates logical block addresses in the tenant's applications to the corresponding physical block addresses in the cloud storage system. The mapping service cannot intervene in every disk I/O operation as it incurs additional latency. Therefore, efficient caching mechanisms are necessary to ensure that the tenants' applications involve the mapping service only occasionally. Additionally, the mapping service needs to maintain persistent data structures to remember the mapping information and to maintain the garbage collection in logical address space. The mapping service faces similar issues like the that of the garbage collection system described above, due to the need to maintain persistent metadata.

1.1.3 Challenges in Cloud Storage's Back-End Management

One of the most important aspects of the back-end management in a cloud-scale storage system is to ensure automatic scalability with minimal to none manual intervention. The back-end system should be able to add/remove various storage components at run-time without disrupting any running disk I/O activity in the entire storage system. Another important aspect of the back-end management is to ensure high availability. The back-end management should provision state-of-the-art technologies to provide software and hard-

ware support in maintaining zero tolerance to data loss. On a cloud-scale storage system, thousands of storage components fail or crash at any point in time and the back-end management has to maintain adequate backups and ensure quick restoration of the failed components. We prefer to focus on JBOJBOD rather than NAS/SAN approach in the back-end because of the following reasons:

- JBOJBOD setup avoids expensive hardware resources like RAID controllers and enterprise-class network switches, and hence the cost factor is significantly high, especially on a large scale storage system in the range of several peta bytes.
- Though JBOJBOD lacks the high-end hardware support made available in an enterprise storage setup, the lack of such features could be made-up using sophisticated software techniques like high availability and software RAID to name a few.
- JBOJBOD setup is very flexible in adding/removing individual disks, to either handle scalability or to replace failed disks. A JBOD device supports disks of different types and sizes, and that offers a huge advantage in a large scale setup, where disks fail at unpredictable times, and the replacement disks are often non-identical to the failed disks.
- A JBOJBOD setup consists of multiple JBOD servers, where each JBOD server is a simple X86 system consisting of a commodity CPU and a set of JBOD arrays attached directly to the CPU. Unlike a NAS/SAN device, JBOD server doesn't appear like a black-box, and hence it is possible to install and run custom-built software on the JBOD server, where the software could be used to either collect some statistics as close to the disk I/O level and/or provide additional functionalities like software RAID.

While it is imperative to ensure flexibility in scaling the hardware resources of a storage system, it is equally important to ensure good overall performance through efficient software techniques. From the software perspective, it is highly desirable to have an efficient software defined distributed storage (SDDS) system that offers storage virtualization by decoupling the hardware functionalities of JBOJBOD from user visibility. The SDDS system is expected to provide automatic management of scalability, reliability, persistency, and other distributed storage related functionalities without letting the user applications worry about the detailed technicalities of the underlying hardware. However, an SDDS system faces several challenges in providing these storage functionalities, and the challenges are better explained when the architecture of the SDDS system is clearly defined in the following section.

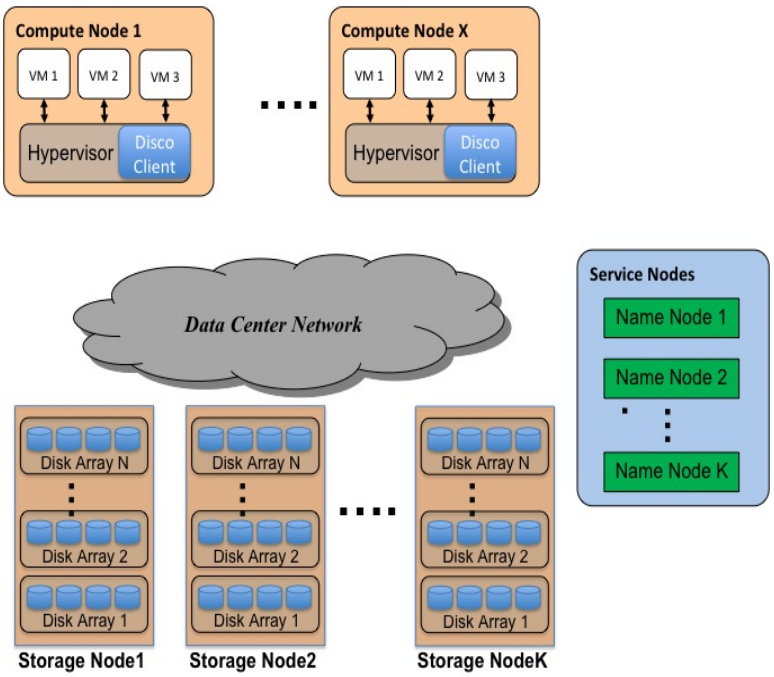

Figure 1.1: *Overview of SDDS architecture consisting of Compute Nodes, Service Nodes and Storage Nodes*

1.2 SDDS System Architecture

The SDDS system segregates the hardware resources into 3 parts: compute nodes (CN), service nodes and storage nodes (SN) as shown in figure 1.1. CNs represent a large set of physical machines, each of which hosts several virtual machines (VM). Tenants who wish to run their applications, can do so, by configuring a virtual datacenter (VDC) with sufficient CPU, storage, memory, networking and other necessary hardware resources. A VDC consists of one or more VMs that are spread across various physical machines. Each VM can again be configured with specific hardware resources similar to that of a VDC, subject to conditions that the sum total of hardware resources on every VM in a VDC doesn't exceed the hardware resources reserved by that VDC. The SDDS system is responsible for managing all the storage related requirements of the VDCs. The SDDS system provides virtual disks (VD) in each VM to allow tenants' applications to manage the storage requirements in a VM, such that the VD appears exactly like a physical disk to the tenants' applications. The SDDS system is responsible for managing storage virtualization between virtual disks and physical disks located in the JBOJBOD system.

The SNs represent the JBOJBOD system and consists of a large number of commodity JBOD servers that aren't arranged in any specific order. Each SN is essentially a commodity JBOD server consisting of a x86 CPU and a set of disk arrays (DA), where each DA consists of a set of hard disks. A DA is just a bunch of disks that makes each disk to be accessible independently. Depending on the specific needs of the tenants, a DA could either be configured with a specific software RAID level setting or be left alone as individual disks.

The service nodes represent a set of physical machines that collectively run applications to manage data flow between the CNs and SNs, and also provide high availability and reliability to the data stored on SNs. The service nodes only help maintain metadata corresponding to the data flow between CNs and SNs, and doesn't actively interfere during the actual data flow. Service nodes are simple x86 servers and we inter-changeably address the service nodes as name nodes (NN) because of the similar functionalities provided by a "name node" in the popular Hadoop [3] literature. The NNs provide two basic functionalities, namely, distributed primary storage (DMS) and distributed secondary storage (DSS). DMS provides iSCSI like block-level storage service to VMs, and DSS helps in backing-up data from primary storage to secondary storage at periodical intervals. Since DMS and DSS operate on a common set of data for majority of its operations, NN integrates DMS and DSS to avoid expensive synchronization delays in handling the common data set, and we refer to the combination as DISCO, which stands for "Distributed Integrated Storage with Comprehensive data prOtection". The following subsections give an overview of the services offered by DISCO.

1.2.1 Block Address Namespace

There are three levels of addressing namespaces in DISCO, namely, logical address, physical address and real address as illustrated in figure 1.2. Blocks in the logical address namespace correspond to the offsets in the VD, as seen by a tenant's application. It is of the form $\langle VolumeID, LogicalBlockID \rangle$, where VolumeID corresponds to the unique volume (interchangeably addressed as VD) mounted in its VM, and LogicalBlockID corresponds to the disk block offset in the volume corresponding to VolumeID. When an application on a VM modifies data in its VD, the DMS allocates new blocks similar to a log structured file system, in order to optimize the write throughput of the storage system. The unused blocks cannot be garbage collected immediately because it could potentially be still in active usage due to the following two reasons: a) volume cloning technique in DSS avoids redundant copy of duplicate blocks by increasing reference count of the corresponding duplicate blocks, and b) deduplication technique (discussed in chapter 3) decreases the reference count

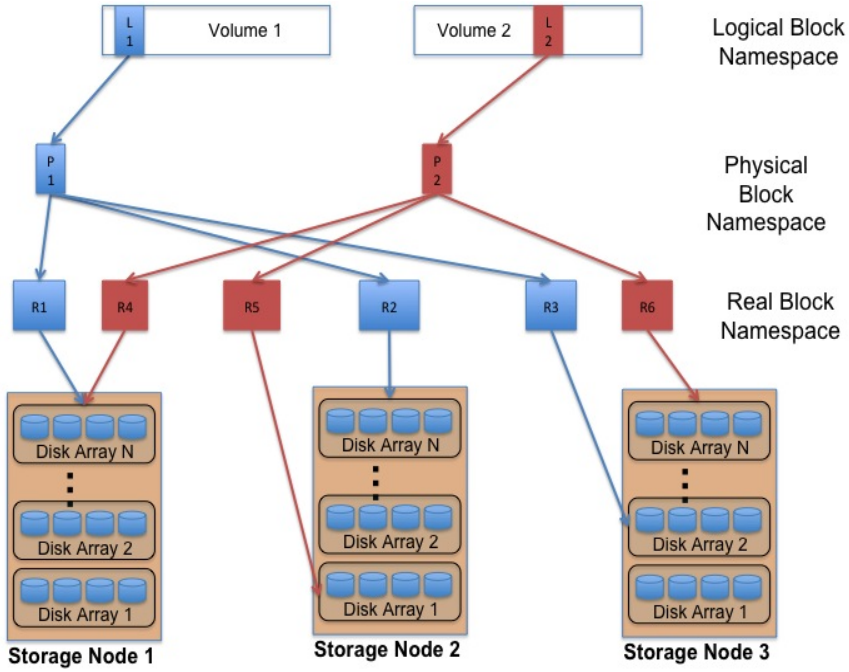

Figure 1.2: *Overview of three level addressing namespace in DISCO*

of those blocks that had a valid reference in the previous snapshot and are no longer referenced in the current snapshot, and increases the reference count of those blocks that serve as duplicates to some block in the input workload. Hence it is necessary to maintain an additional addressing namespace to map every block in logical address namespace to a block in physical address namespace on a M:1 mapping. To ensure high availability, one of the techniques adopted by DSS is to ensure every block in the physical address namespace is replicated on different SNs. The replication functionality necessitates an additional addressing namespace so that every block in the physical address namespace is mapped to multiple blocks in the real address namespace. Blocks in the real address namespace are the disk blocks located in DAs, within their respective SNs, at the lowest addressable I/O granularity. It is of the form $\langle StorageNodeID, DiskArrayID, RealBlockID, Offset \rangle$, where StorageNodeID uniquely addresses each SN, DiskArrayID uniquely addresses a DA within the SN, RealBlockID uniquely addresses the disk block offset within the DA and the Offset uniquely addresses the exact byte location within the disk block. Figure 1.2 shows an example that maps different blocks in each addressing namespace.

1.2.2 Data Path Management

When a tenant's application submits a disk I/O to a VD mounted in its VM, a virtual device driver located in the hypervisor of the CN captures the I/O requests. The virtual device driver is a small DISCO client program that maintains a cache of mappings between the blocks in logical address space to target blocks in real address space. For an incoming I/O request, if the DISCO client program finds a matching cache entry, it uses the corresponding real addresses, or if it doesn't find a matching cache entry, it requests the DISCO server program on NN to fetch the corresponding real addresses. The DISCO server program maintains the mappings between logical address to physical address and between physical address to real addresses, for all blocks in the SDDS system, in a persistent manner. For a read I/O request, the DISCO server program either returns an existing mapping entry between the logical to real addresses for the given block, or returns an error status indicating the address was never used before. For a write I/O request, the DISCO server program always creates a new physical address and corresponding real addresses, and returns the real addresses to the DISCO client program. DISCO implements several optimizations like prefetching and efficient caching techniques to decrease the DISCO server's interference in the data path.

Once the DISCO client program has all the real addresses for a given block, it forwards the I/O request to either all the real addresses in case of a write I/O request. or to a single real address in case of a read I/O request. For read I/O requests, the DISCO client program employs an intelligent read load balancing algorithm (discussed in Chapter 7) to select the least loaded DA among the available options. When a SN receives a disk I/O request, it forwards the request to the corresponding disk scheduler for that DA. Once the disk scheduler processes the I/O request, I/O acknowledgement is returned to the corresponding CN. The DISCO client program then forwards the acknowledgement to the corresponding VD, which is then forwarded to the corresponding tenant's application.

1.2.3 Comprehensive Data Protection

DISCO provides comprehensive data protection over the entire SDDS system at various granularities as described below:

- To protect against disk failures, DISCO uses software RAID with either RAID5 or RAID10 configuration in each DA.
- To protect against disk controller failure, entire DA failure, or networking switch/router failure, DISCO distributes a disk block across the SNs us-

ing one of the standard replication strategies like N-way replication, parity based replication, or erasure codes. With N-way replication, DISCO distributes the data blocks to N DAs, such that all the N DAs belong to N different SNs. Parity based replication is similar to RAID5, where DISCO distributes the data and parity blocks over 4 DAs, such that all 4 DAs belongs to different SNs. RAID5 is just an example, but any of the parity based RAID settings could be employed. With erasure codes, DISCO recodes the data block with K fixed-size sub-blocks. into N sub-blocks, such that N > K and rebuilds the original data block using any M of the N sub-blocks. These are the three most popular configurations found in commercial cloud storage products these days. For simplicity, in this entire dissertation we use N-way replication as the default replication policy to explain various concepts in the SDDS architecture.

- To protect against large-scale failures like multiple SNs failing due to fire accidents or similar incidents, for every volume that a tenant prefers to backup, DISCO takes snapshots of the entire volume at periodic intervals. DISCO stores the backup data using secondary storage that is maintained at a safer archival location, locally within the cloud data center. Typically, the amount of data modified between successive snapshots are very low and hence as a default policy, DISCO takes incremental snapshots on a daily basis. However, large number of incremental snapshots are disadvantageous for two reasons: 1) Time taken to rebuild a volume is quite high if DISCO has to traverse large number of incremental snapshots in order to find the right state of a disk block. 2) If data is corrupted even on one of the many incremental snapshots, DISCO cannot rebuild the volume successfully. For these two reasons, DISCO uses weekly full-backup snapshots in addition to daily incremental-backup snapshots. DISCO uses state-of-the-art deduplication techniques 3 to reduce the amount of data copy during these backup operations.

- To protect against catastrophic failures resulting in entire data center failure, DISCO provides wide area data backup (WADB), that is similar to the local backup, except that the backup is stored remotely from the data center, at a different geographical location.

- To protect against NN failures, DISCO configures NNs in a master-slave fashion. The master NN logs every incoming request to a logging system and processes all incoming requests locally. The slave NN processes the logged requests asynchronously in the background. The logging system maintains all data persistently and in the event the master NN fails,

DISCO uses various optimizations to quickly make the slave NN assume the role of the master.

1.2.4 Dirty Block Tracker

To support incremental backups, DMS uses a dirty block tracker program to keep track of all the modified physical blocks since the previous incremental/full backup. Since DMS records all updates to a physical block on a DA, a single instance of the dirty block tracker program in the DMS is sufficient to accurately track all the physical block-level modifications. The dirty block tracker maintains a list of logical addresses and their corresponding physical addresses, for each dirty physical-block in every volume. DSS uses this dirty-block list during the backup operation to run deduplication and garbage collection programs, which are explained in detail in chapter 3.

1.3 Challenges Addressed by this Dissertation

In this dissertation, we address a few important challenges in a block-level cloud storage system and in this section we briefly summarize them as described below.

DISCO adopts block-level offline deduplication technique to avoid redundant data copy in the incremental backup operation, which backs-up primary storage data to an archival storage. Typical deduplication techniques often bottleneck on disk I/Os, due to the inability to scale main-memory (RAM) resources in proportion to the size of the disk storage space. In a large-scale storage system, disk storage space ranges in the scale of several peta bytes and available RAM resources are typically in the range of few gigabytes. A deduplication system typically maintains an index to lookup for possible duplicates, and due to the mismatch in RAM memory space and disk storage space, most of the lookups result in an expensive disk I/O. For example, on a peta-byte size storage system, if each entry in the index structure occupies 16 bytes to store the hash value and the corresponding physical address of a 8KB block, then the total size of the index is 2TB. This clearly doesn't fit in the main memory. Fortunately, only a few entries in the index participate actively in the deduplication process. Therefore, it is very important to ensure that the index structure either doesn't store unwanted entries or replaces unwanted entries with useful entries in a very efficient manner. It is a touch challenge, to accurately identify and eliminate unwanted entries in the index, and simultaneously ensure that such a technique doesn't significantly reduce the chances of finding a duplicate in the incoming workload. Various solutions

are proposed in previous research efforts, to maintain the most useful part of the index in memory and store the rest of the index on disk. However, the incoming workload has no disk access locality, because the fingerprinting technique induces high degree of randomness. Hence, even if the index structure is intelligently managed to resolve most of the lookup queries in-memory, the occasional disk I/O lookups result in completely random disk I/O, and are sufficient to bottleneck the entire deduplication system, if intelligent prefetching techniques are not used while loading fingerprints from disk into main memory. We built *Sungem* to specifically handle these challenges.

A physical block in the storage system is referenced by multiple blocks due to various events like deduplication, snapshot creation, snapshot deletion and volume cloning. Therefore, it is quite tricky to identify as to when a physical block is free of all references, and when can it be garbage collected, so that the physical block could be made available in the free storage space. A garbage collector either needs to inspect every physical block after each of the events that could possibly modify the reference to a physical block, or should scan the entire storage system at periodic intervals to identify unused block. Either of these approaches are time-consuming and impractical on a cloud-scale storage system. In order to ensure scalability in the range of peta-bytes, the most important challenge for the garbage collector is to spend its efforts in proportion to the size of the modifications in any event. For example, if an incremental backup snapshot is to be performed on a 1TB disk volume, it is generally expected that around 5% of the disk volume is modified with respect to its previous incremental snapshot. Therefore, it makes sense for the garbage collector to work on just the 50GB of modified data, rather than the entire 1TB data. Unfortunately, the current techniques using mark-and-sweep, reference-counting, expiration-time based approaches do not scale well as expected, and hence consume large number of physical resources to do unnecessary additional work. In this work, we integrate the first truly scalable garbage collector with *Sungem*, that scales perfectly in proportion to the size of the modified data, rather than that of the entire volume.

A deduplication system has several tunables, that when accurately fine-tuned, identifies maximum duplicates in the incoming workload, while simultaneously ensuring high throughput. However, fine-tuning requires specific hints about the nature of the workload. Assuming that the workload on a given storage system doesn't deviate drastically, with a detailed understanding of the different data sharing patterns, one could fine-tune their deduplication algorithm accordingly. For example, if certain file types doesn't yield large number of duplicates, then such files can be possibly avoided in the deduplication workload. As another example, a bloom filter incurs multiple hash

lookups compared to a single hash lookup in the fingerprint index. Hence, it is interesting to know if bloom-filter improves or degrades the throughput of a deduplication system. It is possible to answer such questions only if we understand about how frequently does a data block recur in the workload and how many fingerprint comparisons are avoided when a BF is used. In this work, we did a thorough analysis of a real-world data workload, and with a detailed characterization and analysis of the workload, we demonstrated how *Sungem* could be fine-tuned to extract better overall performance.

A disk update request retrieves a disk block, modifies it and writes the resulting block back to the disk. A workload consisting predominantly of such disk update requests, with large working set and low locality, is quite common in a large-scale storage system. Such a workload poses two important challenges that are not convincingly handled by any known storage system, till today. First, a standard technique to mitigate the random disk I/Os in a workload is to buffer the requests in memory and later asynchronously commit the buffered requests to disk in a sequential manner. However, when multiple applications submit update requests, a generic buffering technique is incapable of applying the updates at commit-time, because the update logic is different for each application and the application context is missing at the time of the committing the updates to disk. Second, apart from the low locality aspect of the workload, a disk update request is also challenging because it is fundamentally a disk read followed by a disk write. With the primary motive of delivering high throughput for a workload, a typical storage system services the disk write requests asynchronously using standard buffering techniques, but in order to minimize the response time for disk read requests, it services the disk read requests, as synchronously as possible. Traditional storage systems cannot distinguish between standalone disk reads and disk reads associated with disk updates, and hence the disk updates are treated synchronously, leading to slow disk I/O throughput. In this work, we propose a new storage system architecture called BOSC, to effectively address these two specific issues.

Storage systems use large number of in-memory data structures and hence in order to ensure data persistency, it is a common practice to use disk logging techniques. Since an update to the in-memory data structure doesn't receive completion acknowledgement until the request is logged in the disk-logging system, it is not only sufficient to ensure high throughput in the disk logging system, but it is also necessary to bound the average latency of each request to a very low value. Typically, the size of each update request is in the order of a few bytes and most often it is lesser than the size of a sector on disk. Traditional disk logging techniques often fail to successfully handle such a workload and hence use expensive hardware like NVRAM, to protect the data

from memory crash. In this work, we propose a novel disk logging architecture, *Beluga*, that efficiently manages a high throughput, low latency disk logging system using commodity SATA disks.

Tenants wish to bind performance of VDs with quality of service (QoS) specifications, so that they get a consistent storage performance in a shared data environment, like a cloud storage system. But a QoS specification requires tenants to precisely configure various parameters like bandwidth and latency, which the tenants often find it difficult to precisely define. Even if the tenants are able to define their performance requirements, guaranteeing QoS specifications on an SDDS system is extremely challenging because of the following reasons: First, it is difficult to map application level requirements (ex. 1000 objects / second) to system level requirements (ex. 100 Mega Bytes / second) due to multiple unpredictable factors in the input workload, like I/O request size, workload locality and read/write type of the I/O request. Second, it is difficult to accurately predict the performance of any given I/O request within a single storage device due to sophisticated techniques like NCQ and zoning used in the modern-day hard disks. Third, due to N-way replication, a stream of disk I/Os from a VD are split into multiple sub-streams targeted to different DAs. Thousands of VDs and DAs are present in an SDDS system and an overloaded DA leads to failure in ensuring QoS guarantees. Therefore, it is extremely challenging to balance the loads across all DAs. Fourth, in a massively interconnected SDDS system, it is difficult to accurately identify, isolate and fix the components(VDs, SNs or DAs) that fail to enforce the QoS guarantees. We therefore built a QoS model called *Cheetah*, that uses some novel techniques to enable the SDDS system to provide storage virtualization with accurate QoS guarantees.

1.3.1 What this Dissertation is Not About

In the above sections we gave a high level overview of the cloud storage system and described several challenges in its design and implementation. A cloud storage system consists of several critical components and each pose unique and interesting challenges, and clearly there is no one solution that fits into all the requirements. It is very easy to lose focus on the specific issues that we address in this dissertation and hence the readers are advised that this dissertation is *not* about building a complete cloud storage system. It neither proposes a complete solution to build an SDDS system nor implements the DISCO functionality. The SDDS and DISCO systems described above are used as an umbrella project to highlight the specific research contributions we make in this dissertation. In this dissertation, we have identified a select

few important issues in a cloud storage system and have proposed few novel solutions to resolve them.

Another important aspect is with the use of hardware resources. Our main intention is to use commodity hardware resources and build state-of-the-art software techniques to address few important research problems that we describe in the next section. Therefore, we deliberately avoid using expensive hardware like flash-based SSDs and NVRAM, which is in most of the times a work-around to temporarily fix an issue, rather than solving the underlying fundamental problem. However, we do analyze the relative merits and demerits with such hardware resources whenever we propose to use alternative inexpensive hardware resources. In most of the cases, we use commodity CPUs and 7200 RPM Sata disks in our evaluations to ensure that our proposed research contributions are easily adopted by any storage system using existing inexpensive hardware. Our hope is that our proposed research contributions are useful in designing and implementing a cloud storage system efficiently, such that, in the future, more and more data is moved to the cloud without worrying about typical performance issues that we have already addressed.

1.4 Research Contributions

In this dissertation, we propose several solutions and their implementation techniques, that effectively addresses a few challenging issues as described in the previous Section 1.3. In the process, we make the following novel research contributions:

Deduplication:

- A scalable block-level offline deduplication engine, that delivers consistent high throughput across all ranges of deduplication ratios and improves the deduplication throughput by up to 40% without sacrificing the deduplication ratio, when compared with the state-of-the-art sparse-indexing scheme [6] running with the same amount of RAM, for incremental backup operations,
- The first known garbage collection algorithm whose book-keeping operations are distributed over individual backup operations and which is scalable in the sense that its bookkeeping overhead for each backup operation is proportional to the change to a disk volume between consecutive backups rather than the volume itself,

- A study of two parallelization strategies for data deduplication and garbage collection, in terms of the trade-off between communications overhead and amount of redundant disk I/O,

Trace Analysis:

- A novel user-level dirty-block tracker, which uses the standard file system API to collect file-level changes and derives an incremental block-level backup trace from these changes,
- An in-depth characterization and analysis of a real-world trace that provides unique insights to the dynamics and caveats of modern deduplication algorithms, including the relative merits and demerits of applying a bloom filter to disk data deduplication.

Random Updates to On-Disk Data Structures:

- A new disk access interface that supports disk update as a first-class primitive and enables the specification of application-specific callback functions to be invoked by the underlying storage system,
- A highly efficient storage system architecture that effectively commits pending update requests in a batched fashion and drastically improves the physical disk access efficiency by using mostly *sequential* disk I/O to bring in the requests' target disk blocks,
- A complete prototype implementation of the BOSC architecture and a comprehensive evaluation of this prototype by measuring and analyzing the performance results taken on a BOSC-based B^+ tree, BOSC-based hash table, and a couple of real-world data workloads,
- An empirical demonstration of the efficiency of BOSC, to show that the update request throughput of a BOSC-based B^+ tree implementation is more than an order of magnitude faster than that of a vanilla B^+ tree built on top of the conventional disk access interface.

Fast Logging:

- A logging API that supports fine-grained logging (i.e. logging payload size is smaller than a disk sector) with minimum metadata manipulation and data copying,

- A streamlined disk write pipeline that moves fixed-sized disk write requests across various pipeline stages in data path of a disk block request, at a constant rate, while minimizing the pipeline cycle time,
- A low-power sparse-mode logging scheme that achieves low logging latency without requiring disk head position prediction,
- A comprehensive evaluation of a fully operational *Beluga* prototype that uses three commodity 7200 RPM SATA disks, to deliver 1.2 million 256-byte logging operations, while keeping each logging operation's end-to-end latency below 1 msec.

Cloud Storage QoS:

- A mechanism to provide storage virtualization at both VD and VDC level of granularity.
- A QoS extraction algorithm that quantizes the abstract user-level QoS requirements into system-level QoS specifications that are easier to comprehend and enforce on the storage devices,
- A decomposing algorithm that effectively decomposes the QoS specification into the corresponding QoS specifications for the replica DAs of that VD or VDC,
- A read load balancing algorithm that periodically computes a load distribution map using a piecemeal multiple iteration technique. The distribution map helps the VDs to distribute their workload among the replica DAs, such that none of the DAs in the SDDS system are overloaded,
- A flow control algorithm that regulates the data flow between CNs to SNs in either QoS aware or QoS unaware manner, and
- A mechanism to effectively isolate the performances of each VD and VDC.

1.5 Dissertation Outline

In Chapter 2, we perform an in-depth survey of previous research efforts related to each focussed research component in this dissertation. We describe such previous research efforts with a primary motive to highlight the salient features in such works, and then we show that a few key issues that we have proposed in this dissertation, are neither proposed previously nor is it trivial

to extend such research works to address it in a convincing manner. In Chapter 3, we discuss the architecture of our deduplication system, *Sungem*, and prove its correctness and efficiency through detailed evaluations. In Chapter 4, we characterize and analyze a deduplication trace workload with the primary motive to fine-tune the deduplication system. We do an in-depth analysis to explore various data sharing patterns at both block-level and file-level granularity. In Chapter 5, we describe a new storage system interface, BOSC, that empowers on-disk data structures to efficiently handle update-intensive random disk I/O workloads and prove its correctness and efficiency through detailed evaluations. In Chapter 6, we describe the architecture of a fast disk logging system, *Beluga*, and show how it can be applied to various components in DISCO, along with detailed evaluations to prove its correctness and efficiency. In Chapter 7, we propose several novel techniques to ensure strict adherence to QoS guarantees on the SDDS system, while ensuring maximal hardware resource usage and minimal manual intervention. Finally in Chapter 8, we summarize all our proposed techniques.

Chapter 2

Related Work

2.1 Cloud Storage

Standalone storage systems have been studied over the past several decades and most of the research contributions on standalone storage systems have been successfully applied to cloud storage, as well. However, cloud storage imposes some unique challenges that have gathered lot of interest and attention in the last few years, leading to quite a few interesting research works. Since this dissertation doesn't focus on building a cloud storage system as such, in this section of the survey we give a high-level overview of the design approaches employed by various cloud storage systems.

Cloud Storage techniques differ in several ways depending on the granularity of storage, type of storage devices, disaster recovery mechanisms, and other specific user requirements. Based on the access granularity, the front-end interface of a cloud storage system corresponds to either block-level, object-level, file-level, relational database-level or a key/value store.

With a block-level or object-level access interface, applications outsource just the data storage management to cloud storage systems and the applications focus on transactional-level semantics that are necessary to manage their data. With file-level and other higher levels of access granularity, applications outsource not just the data storage management but also the additional data processing work associated with the data. Such additional data processing tasks include but not limited to the management of file-system, database, transaction-level atomicity, workload partitioning, identifying and removing data redundancies, and these requirements vary for each application.

Amazon's EBS [7] offers data storage at block-level granularity using typical block-level read/write interface in the front-end. The back-end configuration is not explicitly described through publicly available documents. EBS

uses fixed availability-zones to ensure data reliability, where each availability-zone is preconfigured with specific high availability features depending on the geographical location of the data center. Applications requiring additional durability guarantees should do so on their own, and one such possible option is to take snapshots of the entire volume at periodic intervals and store them either on some secondary storage or on a different location on the EBS itself.

Amazon's S3 [8] stores data at object-level granularity, and one of the biggest advantages with such an interface is that the objects are unstructured and could be of any shape and size. It is widely deployed in large number of cloud storage applications because of the generic nature of the object-level interface.

Ceph [9] provides a distributed object storage platform and one of its novel features is to decentralize the data placement process. In traditional distributed storage systems, when an application requests for storage space reservation and requests for the replicas to be placed in a specific manner, the cloud storage system typically chooses a set of least loaded storage servers that satisfy the given constraints. For example, tenants could request the cloud storage system to place X number of replicas on different disks, but on the same rack, Y number of replicas on different racks but on the same storage node, Z number of replicas on different storage nodes but in the same data center. Since the cloud storage system selects one or more storage servers dynamically, it maintains a metadata index that remembers the mapping of logical data block to the corresponding physical disk block located on some storage server. Such metadata indices are usually built upon variants of hash tables or trees, which are persistent on-disk data structures. A representative sample of the index is cached in main memory for quick lookup. Such a metadata index poses several challenges and a few important of them are, a) All data traffic is routed through a set of metadata servers and hence forms a bottleneck in the data access path, leading to single point of failure b) In order to avoid data corruption due to system crash, the metadata index should be logged and that results in expensive hardware to maintain log data and also additional latency in each I/O request. Ceph proposes to avoid metadata index issues completely, by algorithmically deciding the data placement. The data placement logic uses a pseudo-random algorithm that first satisfies the replica placement constraints specified by the tenant and then randomly picks the target locations in the storage cluster allocated to the tenant. However, the tradeoff with such a solution is the lack of flexibility in load balancing the storage system. In spite of sophisticated techniques that predict data workload patterns in advance, it is impossible to make accurate predictions due to high degree of volatility in the data workload. On a storage system that

maintains metadata index, a load balancing algorithm redirects data requests to least loaded servers and ensures overall balance in the CPU, networking and disk bandwidth consumption in the entire cloud storage system. Though Ceph argues that the randomized data placement strategy eventually assures a balanced system, short term fluctuations in workload patterns are predominant in a storage system and Ceph falls short of handling them. GlusterFS [10], Sheepdog [11] and RUSH [12] provide similar algorithmic placement strategies.

Hadoop Distributed File System (HDFS) [3] is primarily designed to provide a truly distributed storage environment that scales infinitely in proportion to the incoming workload. HDFS is the file-system component of a more generic framework called Hadoop [13], and Hadoop is a family of distributed storage protocols that includes functionalities for logging, synchronization, etc. While the front-end of HDFS supports file-level interface, its back-end is a cluster of commodity servers, which are designed for a shared-nothing architecture. The commodity servers are inexpensive systems that perform computation as well as store data locally on that server. HDFS is designed as a shared-nothing architecture because the primary motive is to partition the incoming workload into several sub-workloads and distribute each sub-workload to different servers in the cluster, using map-reduce [14] functionality. Each server is expected to quickly perform the required computations using directly attached disks and return the results to the map-reduce processes. The directly attached storage devices are typically configured using JBODs and the high availability functionalities are typically provided by replicating the data locally within the server using sophisticated software techniques. In some cases, data is also backed-up to secondary storage devices over SAN. HDFS is popular due to its flexibility to run on commodity low-cost hardware without sacrificing on throughput, reliability and availability. However the limitation of binding to specific hardware devices makes HDFS less appealing to be adapted as a generic distributed file system. To be specific, using NAS/SAN devices for primary storage on the computing servers introduces high latencies that are often unacceptable to applications. Another limitation is with centrally managed metadata server, which brings in single point of failure.

Isilon's OneFS [15] is a commercial operating system that is built specifically to offer file-level distributed storage services. One of its advantages is that it can be positioned together with HDFS to enable applications to share data between the computing servers as well as enable usage of enterprise storage products using NAS/SAN technologies [16].

Amazon's RDS [17], provides an entire relational database system (RDBMS) as a service where tenants could adopt a preconfigured database that runs popular RDBMS solutions like Oracle, MySQL and PostgreSQL. A huge advantage

with such a solution is that majority of the administration tasks involved in maintaining a database are handled by the cloud storage system. RDBMS solutions are not easy to scale and if the tenants' requirements are not too strict to match that of a RDBMS solution, then NoSQL solutions are also available on the cloud. Amazon's SimpleDB [18] and Dynamo [19] provides a NoSQL database, which is a simple key value store. They provide seamless scalability, reliability and flexible consistency models that a tenant can pick depending on the specific needs of his applications.

Microsoft's Azure Cloud Storage: [20] offers multiple front-end interfaces through blobs at object-level, tables at RDBMS-level and Message Queues. The front-end interface is provided using the popular HTTP(s) service, where a tenant submits cloud storage requests using web URLs. Back-end is organized as a cluster of storage servers, and each storage server holds peta-byte scale data using multiple racks of hard disks. The storage servers provide fault tolerance through erasure coding and uses two types of replication. On every write at block level granularity, within a storage cluster, data is replicated synchronously to different disks, racks, nodes depending on the high availability configuration, and across the storage clusters, data is asynchronously replicated to two different geographic locations, but the granularity of these replicas are chosen to be much higher to ensure optimal WAN bandwidth usage.

Based on the type of storage devices, all-flash storage systems [21, 22] handle disk-intensive data workloads with better throughput in terms of input-output operations per second (IOPS), compared to storage systems using tapes or magnetic disks [23, 24]. But, such significant speed improvements come at the expense of higher cost. However, with recent trends in large-scale manufacturing and consumption of flash-based SSDs, the cost differences between SSDs and magnetic disks are expected to diminish further in the future. As of today, the per-byte cost advantage with magnetic disks is much higher than for SSDs, but cost alone is not the differentiating factor in deciding between an all-flash storage system and a magnetic disk storage system. SSDs have a definite shelf life that limits the number of writes that each cell in a flash-based SSD can hold. To increase the capacity of a single-level cell (SLC) flash-based SSD, multi-level cell (MLC) flash-based SSDs pack data more densely in each cell and hence the per-byte cost of MLC technology is much lesser than that of a SLC. Additionally, the dense packing of multi-level cells further decreases the shelf life of an SSD. The per-cell write count diminishes from 100K in SLCs to 10K in MLCs [25]. Magnetic disks on the other hand, do not have any advertised shelf-life [26]. Therefore all-flash storage systems are advantageous when a tenant can afford an expensive storage solution and if his application

generates read I/O heavy data workload. SSDs not only deteriorate faster with write I/O heavy data workload, but over time, they also perform poorly with random write I/Os due to the erase-before-write requirement of a SSD. Hybrid storage solutions utilize a combination of magnetic disks and SSDs to reduce the number of writes to an SSD while ensuring majority of the reads are handled by the SSDs [27].

Cloud storage solutions could also be categorized depending on the way the storage devices are connected in the back-end. Some popular technologies include direct attached storage (DAS), network attached storage (NAS), storage area network (SAN) and just a bunch of disks (JBOD). These terminologies are studied in depth over the last several years [28], but JBOD technique is becoming more popular of late, due to its simplistic architecture. Open Vault [29] proposes JBOD array architecture that customizes the JBOD array with minimal processing and other hardware resources, that are just sufficient to manage the data storage. By removing unwanted electronic components in a storage server, Open Vault's design reduces energy footprint, while reducing the overall cost of ownership of the storage server. This open source architecture is adopted by companies like Facebook and Rackspace, and a similar JBOD array architecture is deployed in ITRI's container computer [30] which is the framework that we have adopted in this dissertation, as well.

2.2 Deduplication Techniques

Deduplication is a technique to remove redundant data in a data workload, and there are various deduplication techniques that are used in different contexts of a storage system. In this section of the survey, we give a brief overview of the different classifications of deduplication techniques, discuss some salient features in few important and interesting approaches, and then compare them with our proposed deduplication system called *Sungem*, which is discussed in greater detail in Chapter 3.

2.2.1 Content-Addressable Storage

Content-addressable storage systems (CAS) [31–34] locate data blocks based on the contents of a data block, rather than the logical block number of the data block. Venti [31] pioneered CAS technology and it stores data based on the fingerprint value of a data block. A fingerprint is created by hashing the contents of the data block using standard cryptographic hashes of typically 20 bytes size [35], and the fingerprints are then used for determining the location of the data block. Since the probability of hash collision with a 20 byte fin-

gerprint is equivalent to the probability of data corruption of a data block on disk [31], it suffices if we consider a data block as duplicate even if only its fingerprints match.

A duplicate block is automatically filtered out in Venti, because when an incoming block hashes to the same location as that of its duplicate block, Venti simply uses that location for storing the given block. Venti is designed for a stand-alone storage system and more importantly it is best suited for permanently storing snapshots with indefinite retention period. For incremental snapshots with definite retention period, Venti cannot be used because of its inability to delete a block. HYDRAstor [32] is designed as a distributed backup CAS system with fault tolerance. The Foundation [34] leverages commodity USB external hard drives to archive digital files in a similar fashion to Venti.

2.2.2 Timing of Backup

In the context of data backup systems, deduplication techniques can be categorized according to whether they are designed to handle outputs from a full backup operation or from an incremental backup operation. A full backup operation takes a snapshot of the entire storage volume and then deduplicates while copying the snapshot data to the back-up storage. However, an incremental backup operation feeds on the list of changed blocks between consecutive snapshots and then deduplicates while copying the snapshot data to the back-up storage. The amount of data modification between two consecutive full backup operations is expected to be a small percentage of the volume's size, and hence the snapshots of consecutive full backup operations are expected to overlap significantly. As a result, the number of duplicates in a full backup snapshot are typically very high. Therefore, deduplication techniques vary significantly depending on whether the input workload is from an incremental backup or full backup operation.

As a default policy, DISCO performs weekly full-backups and daily incremental backups for all virtual disks (VD). Since a full backup has lot of duplicates, it is relatively simpler to identify duplicates in a full-backup. However, in an incremental backup, due to lesser number of duplicates and lower data locality, a deduplication system is presented with tougher challenges. Therefore, DISCO primarily targets *Sungem* for incremental backups, but also uses *Sungem* for full-backups to ensure code reusability and easier deduplication system management across all data backup workloads.

2.2.3 Data Comparison Techniques

When a deduplication system receives an object to be deduplicated, it has to determine if it has already seen a similar copy of the object and accordingly reply to the caller whether the object is unique or not. The object under query could be of any size, depending on the granularity of the storage system, and hence the deduplication system has to compare every byte of the given object with every byte in all the possible duplicate object candidates that are recognized by the deduplication algorithm. This is the only way, that a deduplication system can ascertain with 100% confidence if the given object is unique or not. However, such a technique has several problems like, a) It is very expensive to perform large number of byte-by-byte comparisons, especially for large size objects. b) The deduplication system receives large-scale workload and if every comparison involves a disk I/O followed by a large number of byte-by-byte comparisons, the time taken for deduplication far exceeds the window for backup time.

Therefore, in order to avoid the expensive byte-by-byte comparison with all the blocks in the backup archive, fingerprints of the data objects are used for comparison. Since a fingerprint is typically of 20 byte size, a 20 byte comparison is definitely much better than a byte-by-byte comparison on several kilo-bytes to mega-bytes of the object size. Though the probability of collision in a 20-byte fingerprint is expected to be very low, there is still some infinitesimal chance of a collision. When such a collision happens, there is data loss because of incorrect identification of a duplicate. Therefore, some deduplication systems that have zero tolerance to data loss, adopt fingerprint comparisons to identify possible duplicates in the first stage, and then in order to confirm duplicates with 100% assurance, in the second stage, they perform byte-by-byte comparison between the given object and the duplicate candidate object.

2.2.4 Granularity of Deduplication

Based on the granularity of the basic unit of duplicate matching, deduplication techniques could be applied at either file-level, block-level or byte-level. There is no obvious advantage in picking one of these granularities over the other, because it's a tradeoff which depends on the context where deduplication is applied and the operational costs involved.

File-level deduplication [36] techniques use an entire file as the basic unit of deduplication. Files are compared against each other using their fingerprints. When the fingerprints match, the file is marked as duplicate and the associated disk blocks are garbage collected to later release them to the free space, and

all future accesses to the given file are redirected to the duplicate file. The file-system needs to carefully manage the metadata of both the duplicate files because security related information should not be violated at any cost and the file owner should not be aware that his file is being deduplicated. At file-level granularity, though the design is simplistic, the need to modify the file-system in the operating system's kernel, makes it difficult to port the deduplication solution to multiple operating systems. File-level deduplication is not a good fit in systems where files are often modified, rather than being shared without any modification. Even if there is a single byte change between any two files, the two files look completely different and separate copies of the two files are stored in a file-level deduplication solution.

Block level deduplication [32, 34, 37, 38] uses disk blocks as the basic unit of deduplication. Typically, block-level deduplication techniques are applied below the block layer of an operating system, just before a disk block is submitted to a disk. Since this doesn't involve any modification to the operating system, it is easily portable across different operating systems. This approach works very well in cases where file modifications are very common. When two or more large files, differ in only a few bytes and share most of the other data, only the modified blocks are stored in the deduplication engine's database as a new copy, and the rest of the blocks in the files are marked as duplicates. Hence the probability of finding duplicates at block-level is higher than that at file level granularity. But due to the higher number of fingerprint comparisons at block-level, the CPU usage increases accordingly.

Byte level deduplication [39] identifies duplicates at byte level and identifies the maximum possible duplicates compared to block-level and file-level deduplication techniques. Unlike those techniques, a byte-level comparison is much simpler because the maximum possible values of a byte are limited to $(2^8 = 256)$ values and hence a table with counters could be used to match an incoming byte to affirmatively identify a duplicate. However, the biggest disadvantage with this approach is the high CPU consumption. If a large file of several megabytes in size is given for deduplication, a file-level deduplication technique confirms if its a duplicate or not within just one lookup into its database and possibly does a byte-by-byte comparison of all the bytes in its representative fingerprint.

DISCO architecture is better suited for block-level deduplication, and since block-level deduplication solution is portable and results in higher savings in storage space, in this dissertation we discuss block-level deduplication in Chapter 3. We also analyze various tradeoffs between file-level deduplication and block-level deduplication in Chapter 4.

2.2.5 Positioning of Deduplication

Primary Memory vs Secondary Memory Deduplication

Data deduplication techniques could either be applied to the disk blocks on secondary storage or to the operating system (OS) pages on primary memory (RAM). In a virtualized system, multiple guest OS instances run on a host OS and if multiple guest OS's are identical, then most of the OS related data stay unmodified for long time. Since host OS allocates physical memory for guest OS's, common data pages on guest OS's can be shared, making redundant memory available in the free memory pool. RedHat unofficially reported that using linux kernel same page merging, also called as kernel shared memory (KSM), KVM can run as many as 52 Windows XP VM's with 1 GB of RAM each on a server with just 16GB of RAM. KVM scans parts of memory and calculates a hash of entire OS page. It then tries to match identical pages using these hash values. Since most of the OS libraries are unchanged, most of the OS memory is shared. The duplicate pages are marked as free and shared page is write protected with a copy on write flag, so that any future modification to that page, will result in a new page to the host which modified it. Primary memory deduplication competes for CPU with user applications and hence it's usage should be moderated based on user application load. Hence not all pages are scanned for duplicates in OS kernel. For example on a linux kernel, only those marked with madvise system call are considered for scanning. Another interesting candidate for primary memory deduplication are zero pages, since they avoid any comparisons, and help share memory among different guest operating systems in a virtualized environment.

Secondary memory deduplication is applied to disk blocks that are stored on storage devices. The latency restrictions are slightly relaxed compared to the primary primary deduplication technique. These are further classified into on-line and off-line deduplication techniques.

In-line vs Post-process Deduplication

In in-line deduplication, duplicates are eliminated before data is copied from primary storage to backup storage. Since this involves duplicate identification in real-time, it needs to be extremely fast. Usual practice is to store fingerprints of the unique objects in an fingerprint index (FI) and use it to compare against incoming fingerprints for deduplication checks. In order to handle large-scale data volumes, the size of such an FI cannot fit in main-memory. For example, a 2 peta-byte data volume with 8KB block size and 20 byte fingerprint size needs a maximum of 5 Tera Bytes FI. Since this FI cannot fit in main-memory, it has to be stored on a persistent storage device like a flash-based SSD or

magnetic disk. But it is impractical to involve a disk lookup operation on the FI for every disk write I/O operation handled by the inline deduplication system. Therefore FI management is the most critical aspect of designing any deduplication system. Zhu et al. [37] propose to use an in-memory bloom filter to process a large majority (99%) of the incoming fingerprints without involving expensive disk I/Os. Only when the bloom filter fails to identify the unique fingerprints, a disk I/O is performed to search for possible duplicates on disk. They also propose some intelligent prefetching techniques to amortize the occasional disk I/Os and these are discussed in detail in a later section. ChunkStash [40] proposes to use flash-based SSDs instead of magnetic disks, to store the part of FI that cannot fit in main memory. Since the I/O latency of a flash-based SSD is closer to that of DRAM than to that of a hard disk, the overall throughput of a deduplication process improves when compared to a deduplication system using magnetic disk-based FI. However, flash-based SSDs suffer from several drawbacks, as discussed in earlier sections.

The advantage with inline deduplication is that there is lesser stress on disk controllers as the data storage needs correlate directly with the observed deduplication efficiency. But since this happens at real time, it shares CPU and memory resources with user applications. Hence inline deduplication technique continuously monitors the system load to decide the aggression factor of the deduplication operations.

Post-process deduplication, also typically referred to as off-line deduplication, is performed outside the active disk I/O path, by an asynchronous process, after the data is stored on the disk. Even though this doesn't have the run time requirements like that of the in-line deduplication, it still needs to complete within a short span of time. Since the deduplication operation has a lower priority than the real-time disk I/O activity on primary storage, the timing window of the backup operation is generally chosen such that there is none to limited disk I/O usage on the primary storage. Compared to in-line deduplication, post-process deduplication requires larger primary storage, because duplicate data is removed only after its first saved to the disk. Though the eventual disk space usage is similar in either of the approaches, the disk storage capacity in post-process deduplication cannot be based on the actual unique data on the system. Post-process deduplication solutions [41–43] are most benefited on live storage systems, where the primary concern is, to provide low-latency access to live data, than to bother about the temporary higher storage space utilization. On the the other hand, in-line deduplication systems are most benefited by backup systems where the primary focus is, on backup throughput and disk space utilization, than on the latency of individual backup requests.

Source vs Target Deduplication

To ensure high availability, data is typically backed-up to a secondary storage on a remote location. Data deduplication can be applied before transferring the data over the network, which is called source deduplication, or can be applied at the remote location after transferring the data, which is called target deduplication. In a cloud-scale storage server, multiple storage clusters backup data to a backup server, that is either positioned on a different storage cluster within the same data center or on a storage cluster positioned at a geographically different location. In either cases, backup process involves moving data over the network.

In source deduplication, duplicates are eliminated within each storage cluster and only the unique data is sent over the network to be stored in the backup-server. This saves the precious network bandwidth but utilizes more resources at source node, often competing for resources with other client applications. In target deduplication, all the backup data is transferred over the network to the backup server, where duplicates are identified and removed. This consumes more network bandwidth because redundant duplicate data is transferred over the network. However, the chances of finding duplicates are higher in target deduplication, compared to source deduplication, because multiple storage nodes backup their data to the backup-server and hence a larger set of data is available for comparisons. A combination of both these approaches [44] can be used, provided there is enough time and resource to allocate for the deduplication operation.

DISCO uses a variant of the target deduplication. Instead of sending the entire data to the backup-server, only the representative fingerprints are first transferred to the backup-server, where *Sungem* is installed. Upon deduplication, if a fingerprint is identified as unique, then DISCO transfers the corresponding data block to the backup-server. If a fingerprint is identified as a duplicate, then *Sungem* returns the block-id of the duplicate block and DISCO updates its metadata accordingly to redirect all future disk I/O accesses on the given disk block to the duplicate disk block. DISCO also this information to perform post-process deduplication on primary storage.

2.2.6 Variable Segment vs Fixed Segment

When a deduplication system receives a stream of incoming data objects from an incremental data backup, to be deduplicated, a naive method is to process each object to find its duplicate. Such a naive method is very time consuming and more importantly it is not very common to have every block in a file to be modified during an incremental snapshot. Therefore it is advantageous

to group together consecutive objects in the incoming stream in a content dependent manner and then lookup for a duplicate of only the first object in the group. Deciding on the size of a group is not straightforward because it is a tradeoff between CPU usage and finding more duplicates. A larger group size reduces the number of fingerprint comparisons at the expense of lowering the probability of finding duplicates. The grouping technique is commonly referred to as chunking and chunking could be done either with fixed size or variable size.

In fixed size chunking technique, a fixed number of fingerprints are chunked into a segment and the deduplication system is presented with a stream of such segments. Fixed size chunking is easier to implement, and is best suited to applications where file-level deduplication is beneficial, because in such a workload, file-level modifications are expected to be minimal and hence most of the fixed-size segments are identical. Foundation [34] and rsync [45] use fixed size chunking technique because of its simplicity.

If the modifications to a data object result in even a slight shift of data in the segment, a fixed-size chunking technique that uses only the head of the segment for locating the duplicates, fails to identify the whole segment as a duplicate. Variable-size chunking technique partitions the incoming stream of objects at multiple anchor points, in such a way that the segment size is shrunk or expanded to match the duplicate segments. Variable size chunking is most efficiently implemented using Rabin fingerprinting algorithm [46] based sliding window hashes to identify segment boundaries. It is employed by several deduplication systems [32, 37, 38, 47]. Bimodal Chunking [48] further optimizes variable length chunking. It uses smaller chunks in limited regions of transition from duplicate to non-duplicate data, and elsewhere it uses larger chunks. Variable-size chunking is more sophisticated because of the way it manages multiple anchor points to identify appropriate segment sizes and hence identifies more duplicates than a fixed-size chunking technique. However, for the very same reason the variable-size chunking technique is time-consuming and results in increased consumption of CPU cycles.

Sungem uses a fixed-size chunking technique to partition the incoming stream of fingerprints into segments of 256 fingerprints, and uses a novel sampling technique to sample the segments stored in its database at various anchor points, so that even if a fixed size incoming segment contains a sequence of duplicate fingerprints at some offset within the segment, the sampled FI successfully identifies the duplicates with the same accuracy of a variable size chunking technique. Therefore, *Sungem* does chunking as fast as fixed size chunking and identifies duplicates as high as a variable size chunking technique.

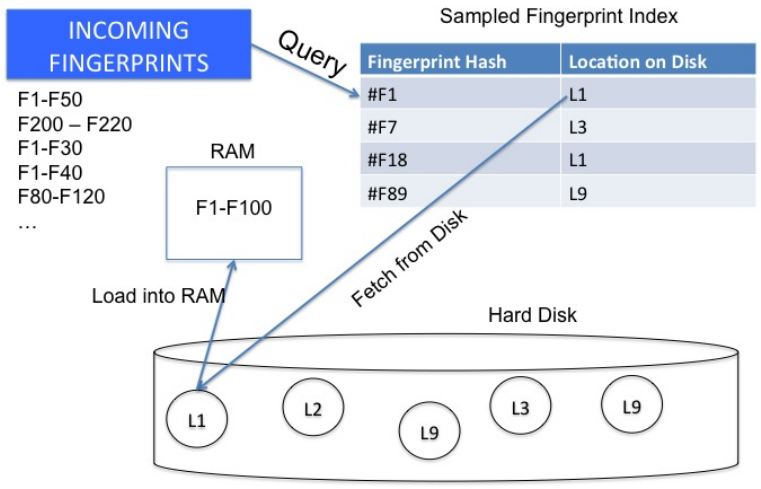

Figure 2.1: *Fingerprint sampling and prefetching used in modern disk data deduplication*

2.2.7 Faster Index Lookup Strategies

On a large-scale storage system, as discussed earlier in Section 2.2.5, if the FI has to hold fingerprints for all blocks in the storage system, then FI has to be stored on disk, because such a large size FI cannot fit completely in main memory. Additionally, there is hardly any spatial locality among neighboring fingerprints in the input workload because fingerprints are generated by cryptographic hash functions, whose primary motive is to ensure good randomness. As a result, in the worst-case, every look-up into the FI could result in at least one disk I/O, and that drastically slows down the deduplication performance, and hence affects the data backup process. Therefore, index lookup management is critical to the deduplication system and many previous research efforts have focussed primarily on optimizing the index-lookup operation. Such optimization techniques can be broadly classified as *prefetching* and *sampling*. Prefetching is a technique used to load a group of fingerprints before-hand, in anticipation that they would be used very soon in the future. Sampling is a technique used to filter out unwanted entries in FI and store only the necessary fingerprints that are expected to be referred more than once. The decision to choose what is necessary and unwanted, is very critical to the performance of the deduplication system. Though, one wants to identify them accurately, its impossible realistically and hence the onus is to minimize the negative impact of the sampling heuristics on the deduplication performance. The sampling and prefetching scheme is illustrated in figure 2.1. It can be seen that the sampled fingerprint index table holds only a few important fingerprints. When an

incoming stream of fingerprints in the range $\langle F1 - F50 \rangle$ is queried, only F1 is found in the sampled FI. Instead of loading just F1, the prefetching scheme loads the entire set of fingerprints in the range $\langle F1 - F100 \rangle$ to cache memory. Such a prefetching scheme, helps find duplicates for the fingerprints $\langle F2 - F50 \rangle$ without incurring additional disk I/Os.

As discussed previously, Venti [31] pioneered CAS technique and proposes to use fingerprints of a data block as the target location of that block on the disk. The focus is more on ensuring write-once-read-many property, rather than to optimize the deduplication property. Venti has a slow disk I/O performance because fingerprint values of adjacent data blocks in the incoming workload have no correlation with each other, because of which the locality in the disk I/O is very low. Though, Venti proposes to use a combination of block cache, index cache and disk striping to improve the write throughput, even after using 8 high end Cheetah SCSI disks for striping, the final write throughput is a mere 6.5 MB/Sec. The disk I/O throughput is so low because of the low locality and low cache hit ratio. Therefore, it suggests that caching will only marginally improve the random disk I/O throughput, and that could have serious consequences on a large-scale storage system.

Zhu et al. [37] propose an in-line deduplication technique for a disk-to-disk (D2D) [49] data backup system. In order to avoid disk I/O lookups for every FI lookup, they propose three novel techniques; (1) a bloom filter [50] based summary vector, (2) stream-informed segment layout (3) locality-preserved caching. The bloom filter avoids FI lookup for most(99%) of the incoming fingerprints that are guaranteed to not have a duplicate. Since a bloom-filter can have false positives, those fingerprints for which the bloom-filter responds positively, FI is looked up to find a matching duplicate fingerprint. When a disk I/O is used to lookup a fingerprint stored on the disk, rather than loading in just the required fingerprint, the entire container holding the required fingerprint is prefetched in the anticipation that the neighboring fingerprints in the container could be referenced soon. Containers are just an abstraction of the basic unit of I/O operation and they are much larger than a usual disk block, because they are designed to correlate the average segment size. The prefetching advantage is made possible by placing together all the fingerprints in the same segment in the incoming workload, into a single container, and this technique is referred to as stream-informed segment layout. This techniques helped achieve over 100 MBPS throughput and more importantly it demonstrated that disk I/Os could be effectively avoided in the index lookup operation.

The sparse indexing scheme [6] addresses the FI disk bottleneck issue by using sampling the FI to reduce the memory footprint of the FI. The incoming

stream of data objects are first partitioned using fixed-size chunking and then again partitions with variable-size chunking technique to aggregate a set of chunks into segments. A sparse-index is used to identify which stored-segments contain maximum number of hits to the chunk hashes in the given segment, and identifies such stored-segments as champions. Each champion stored-segment is loaded from disk and the corresponding chunks are compared to identify the duplicates. The stored-segments are organized in a manner similar to Zhu et al. [37]. This scheme allows the storage system to decide on a tradeoff between the probability of finding a duplicate and the amount of sampling. If more samples are used, more duplicates could be found and if lesser samples are used, the chances of missing a duplicate is higher. However, this scheme enforces a fixed sampling rate for all the chunks without differentiating them based on the usefulness or stability of a segment.

ChunkStash [40] proposes a flash-based FI lookup to ensure high throughput deduplication systems that cannot afford even a disk lookup for 1% of the cases that Zhu et al. [37] suggests to lookup in a disk. Since flash-based SSDs can support faster lookups than a magnetic disk, the entire FI is maintained on the flash-based SSDs. Random write I/Os to flash-based SSDs are known to be slow due to erase-cycles issue that we discussed previously. So they propose to overcome it by converting random updates into log structured append operations and then asynchronously commit the data using a sequential sweep on the disk. Their evaluations show that using flash disks with several gigabytes, 200 MB/sec throughput is achieved. Flash-based SSDs are still quite expensive and for the marginal improvements in throughput, it is hard to adopt such a solution.

Fanglu et al [51] suggest progressive sampled indexing. Typically blocks are grouped together into segments and the fingerprint lookup index is sampled to make sure only important fingerprints are represented to cover maximum stored segments and also ensure that entire index fits in main memory. However, they suggest a sampling scheme based on amount of free memory available at the time of sampling rather than using the total main memory capacity. Such a progressive sampling scheme dynamically changes the sampling rate and yields better deduplication rate.

Extreme-Bin [36] deduplicates data based on files in a distributed storage system. Fingerprint Index (FI) is distributed to multiple nodes. Each input file has a sampled fingerprint to route it to a distributed node, and a duplicate block in the input file is detected by consulting the block's fingerprint with the distributed FI. Besides a sampled fingerprint, each file has a whole-file fingerprint. A match of the whole-file fingerprint indicates that the whole file is a duplicate without comparing individual fingerprints in the file.

Our proposed deduplication system, *Sungem*, uses three novel techniques to reduce the disk I/O bottleneck problem. First, *Sungem* adopts the benefits of variable sampling technique into FI, by sampling the FI at various anchor points. Second, to make the most of the memory space reserved for storing fingerprints, *Sungem* varies the sampling rates for fingerprint sequences based on their stability. Third, *Sungem* uses temporal locality rather than the spatial locality to store a segment in a container. In other words, *Sungem* puts *related* fingerprint sequences, rather than fingerprints from the same backup stream, into the same container in order to increase the fingerprint prefetching efficiency. With these techniques, *Sungem* is able to produce an order of magnitude higher throughput than any other deduplication system discussed in the literature, while using only a few commodity, inexpensive SATA disks.

2.2.8 Medium of Backup Storage

Disk I/Os are one of the major controlling factors that decide the overall performance of a deduplication solution. Hence the deduplication techniques customize their algorithms to extract maximum performance from the underlying storage devices, which could be either a tape library, magnetic disk or a flash-based SSD.

Zhu et al. [37] argues on the relative merits of a disk based solution, when compared to a tape based solution. It's argued that sequential data transfer rates on tapes are better than that on disks, but the disadvantage of tapes is with the requirement of manual intervention, which is expensive and error prone. Critical restore operations can fail because of mishandling of tape cartridges, and moreover, random access operations are predominant in restore operations. So tape is not suitable, as it's poor random access performance would seriously bottleneck the entire deduplication operation.

Cost-per-byte is not a major differentiating factor between a tape and a disk, but tape is preferred in cases where data needs to be backed up for a long time. Though disks do not have any advertised shelf-life, they are believed to have a shorter life-time than a tape, and hence disks need to be rotated, which results in higher maintenance cost on the disk. Some backup solutions [52] adopt a combination of disk and tape, to ensure faster recovery operations using disk and inexpensive archival storage using tapes. Contrasting to the expectations, Gartner research [53] reveals that tapes have a surprisingly higher failure rate. It is claimed that typically, only 70% of the times, tapes are accessed from archival storage, and among them, an astonishing 10%-50% of the tape restores fail. Even though, disks are expected to have a higher failure rate than tapes, disks can be made more resilient using high availability techniques like RAID.

There is another interesting technique called virtual tape library (VTL), which virtualizes disk storage as tape libraries, allowing easier integration of VTL with existing backup software, recovery processes and policies. The benefits of such virtualization techniques include storage consolidation and faster data restore processes. TLFS [54] demonstrates the use of VTL in a file system and compares it with disk and tape storage solutions.

2.2.9 Deduplication Trace Workload Analysis

In the past several years, many characterization studies on local file system [59–61] and network file systems [62, 63] have been done, which are mostly focussed on file access patterns and file characteristics spread over large periods in time. Results of the analysis are used to design modern file systems and can also help in collecting detailed workload statistics that are targeted explicitly for deduplication workloads.

Primary data workload characterization done by Microsoft [64] uses large scale datasets covering wide range of access types and characterizes the workload based on typical components used in an online deduplication system. In online deduplication, the onus is on minimum usage of hardware resources and hence the workload characterization is focussed heavily to optimize the disk, memory and CPU resource usage. Since a backup data has large number of whole file duplicates, it differs very much from a primary data workload and hence the characterization of such workloads differs very much from the primary data workload.

Studies on backup data characterization vary from smaller dataset analysis [65] to very large data sets analysis [66, 67], but they all focus on identifying file system trends related to backup storage. They highlight the core differences between backup and primary data workloads and characterize the backup workload to show the specific qualities in a backup workload that helps design better deduplication technologies. Specifically, [65] addresses typical bottlenecks in deduplicating a full backup and their work is closest to our work, yet differs significantly because of two reasons. First, the characterizations used in their paper are well understood today and a more detailed analysis is required to target todays deduplication techniques. Second, In our work, we thoroughly analyze various duplicity patterns in the workload and to the best of our knowledge, this has never been discussed before in this literature.

2.3 Garbage Collection Techniques for Deduplication Storage Systems

To simplify the comparison study between various garbage collection (GC) algorithms, let us use a reference backup system with the following configurations and assumptions. The reference backup system has 1PB worth of physical blocks with 8KB block size and supports four 32TB VDs, each of which is fully utilized. Assume one backup is taken for each VD every day and each backup snapshot is kept for 32 days and then discarded. Also assume that every block gets accessed 64 times before it's evicted. Though the last assumption is somewhat arbitrary, we made them up only to quantitatively compare the *relative* performance of the GC algorithms we consider below. In order to represent an incremental backup snapshot, the backup system maintains a logical to physical translation (L2P) map, where the logical addresses correspond to the logical disk blocks in the VD and the physical addresses correspond to the target location of those disk blocks on the respective storage devices. In addition, the garbage collector maintains a *physical block array* (P-array) that maintains metadata for each physical block in the entire storage system. Therefore, at any point in time, there are totally 128 backup snapshots in this system and the percentage of change between consecutive incremental backup snapshots of a VD is assumed to be 5% of the VD's size.

GC Schemes	***Lookup cost***
Mark and Sweep	512 Billion
Ref Count	16 Billion
Expiry Time	8 Billion
Hybrid RC/ET	0.4 Billion

Table 2.1: *Comparison of the lookup cost overheads for four GC algorithms using a reference data backup system whose detailed configuration is described in the text.*

2.3.1 Mark and Sweep

A naive mark and sweep approach [68] scans the L2P maps of all active backup snapshots for all VDs, and marks only those physical blocks that are actively referenced. Upon completion of the mark phase, the sweep phase begins wherein all the non-marked blocks are garbage collected. For the reference backup system, one needs to lookup $128 * \frac{32TB}{8KB} = 512$ Billion L2P map entries

in a largely sequential fashion. The P-array needs to accommodate $\frac{1PB}{8KB} = 16T$ entries, where each entry is represented by a 1-bit flag to mark the presence of a physical block. Therefore, the total storage space required by P-array is $\frac{16T}{8} = 2TBytes$. Clearly the main memory cannot completely accommodate either of these structures and hence a large majority of the entries have to be stored on the disk. Such a naive mark and sweep GC algorithm has the following major drawbacks:

1. Even though the accesses to both the L2P map and P-array are largely sequential, since both these structures are stored mostly on disk, the large number of disk access requests result in very low overall throughput of the GC process.

2. For the overall duration of the mark and sweep phases, the entire VD has to be frozen, or else a block referenced during an ongoing mark phase could potentially miss being captured and the sweep phase could garbage collect such an active block, leading to data corruption.

While 1) results in large delays, 2) leads to long pause times, either of which hurts the overall GC performance. Hence the naive mark and sweep approach is impractical for a large scale storage system.

HYDRAstor [32] employs a variation of the mark and sweep GC technique, where instead of freezing the entire system from doing any I/O activity on any of the VDs, all the VDs are marked read-only. However, the *m*ark phase can still be prohibitively long if the VDs are dominated by write I/Os. Fanglu et al. [69] propose group mark and sweep (GMS) mechanism, whose key idea is to avoid touching every file in the mark phase and every container in the sweep phase to make GC scalable and fast. However, the GMS technique operates at the file-system level to track modified files and hence groups a set of modified files to perform mark and sweep on selected areas in the storage system.

2.3.2 Reference Count based

The simplest example of the local metadata bookkeeping approach is *reference counting* [70–72], which maintains a reference count for each physical block to record the number of backup snapshots that point to it. When a backup snapshot of a VD is taken, the reference count of every physical block the snapshot references is incremented. When a backup snapshot of a VD is retired, the reference count of every physical block the snapshot references is decremented. When a physical block's reference count reaches 0, it is collected and put in the free pool. Assuming each P-array entry keeps a 2-byte reference count, the number of lookups in the P-array is $\frac{32TB}{8KB} * \frac{1}{64} * 128 * 2 = 16Billion$,

where a factor of 2 is multiplied because reference count of every block is updated both at creation and deletion times of a snapshot, and the factor $\frac{1}{64}$ refers to the assumed degree of reuse for every fetched block. We account for all 128 snapshots because we are comparing with mark and sweep approach which can be scheduled to run after aggregating multiple snapshot creation and deletion events. Although the number of lookups are much lesser than the mark and sweep approach, updating 16 billion entries with random locality disk IO accesses will obviously cause the system to bottleneck.

2.3.3 Expiry Time based

The retention period of a VD is configured at the time when the backup snapshot is created, and since its known beforehand, it is possible to determine the last moment at which a backup snapshot continues to reference a physical block. Suppose a backup snapshot is created at time T and its retention period is R, then this snapshot will not reference any of the physical blocks it references after $T + R$. Assume we maintain an expiration time for every physical block, which indicates the time after which the block can be freed. When a backup snapshot of a VD is taken, the expiration time of every physical block the backup snapshot references is set to the larger of the current expiration time and the current time plus the snapshot's retention period. With this arrangement, no additional actions need to be taken when a backup snapshot of a VD is retired. To reclaim garbage blocks, one scans the P-array, each entry of which in this case maintains a 2-byte expiration time, and those physical blocks whose expiration time is less than the current time are garbage blocks.

Unlike reference-count based garbage collectors, expiration-time based garbage collectors [73], cannot immediately reclaim a physical block that is no longer referenced by any logical block, but instead have to wait to garbage collect, until the expiration time of a block expires. As a result, a key advantage of the expiration time-based scheme over the reference count-based scheme is that no actions need to be taken at the time when a backup snapshot is retired. An asynchronous scanning process can be scheduled at any time after a snapshot expires to reclaim all the expired blocks. Therefore, for the reference backup system, the total number of lookups in the P-array, required to create and retire backup snapshots at the end of each day is $\frac{32TB}{8KB} * \frac{1}{64} * 128 = 8$ Billion. The factors in this equation are very similar to those in the reference counting approach except that we no longer need to account for any action when the snapshot is retired. Hence the number of lookups in the expiry time based approach is half of that in the reference count approach. However, a limitation of this scheme is that the retention period of a backup snapshot cannot be modified after the snapshot is taken.

2.3.4 Summary of GC comparisons

Table 2.1 shows a detailed comparison among the four GC algorithms discussed in this section. In the first approach, batched GC algorithms such as *mark and sweep* run periodically, require system pause, touch a fixed amount of metadata in each activation that is independent of the interval time between successive mark and sweep invocations, and incur largely sequential disk accesses for a huge number of P-array and L2P map accesses. In the second approach, incremental GC algorithms such as *reference count* and *expiration time*, run incrementally, do not require system pause, touch an amount of metadata within a time interval that grows with the interval's length, and incur largely random disk accesses.

Consequently, we propose a hybrid approach, where *Sungem* takes the second approach, which incurs run-time performance overhead due to metadata bookkeeping. To minimize this metadata bookkeeping overhead, *Sungem* adopts the BOSC (Batched mOdifications with Sequential Commit) mechanism [74] (explained in detail in Chapter 5) to modify the on-disk P-array. The main advantage of the proposed hybrid GC algorithm is that the number of P-array entries that the GC needs to modify is proportional to the number of modified blocks in a input snapshot. Therefore, for the reference backup system, the total number of lookups in the P-array required to manage the snapshots in a given day are: $\frac{32TB}{8KB} * \frac{1}{64} * 128 * 0.05 = 0.4$ Billion. The major factor in this equation that brings down the lookup count is the operation over 5% delta list change instead of the complete list of blocks in a snapshot. The total amount of metadata that the proposed hybrid GC algorithm needs to touch is proportional to the amount of block-level change, and hence it can be easily shown that its total metadata update overhead is no worse than any known *mark and sweep* variants. Therefore, the proposed hybrid GC algorithm is the first known GC algorithm that is both incremental, in terms of not requiring system pause, and minimal, in terms of total metadata update overhead.

2.4 Fast Random Updates to On-Disk Data Structures

A common approach to improving the performance of small disk writes is to temporarily buffer the disk writes to a fast storage medium like NVRAM, and then asynchronously submit the buffered writes to data disks. Such a buffering technique provides two benefits: scheduling disk writes more flexibly and combining multiple writes with the same target. However, NVRAM is expensive,

and for workloads with poor locality, high update rate and large working set such as TPC-C [75], a small amount of NVRAM can only mask the delay for a finite number of disk writes, because eventually the sustained write performance is bottlenecked by the speed at which writes are propagated to disks. Write-only disk cache [76] mitigates the performance problem due to buffer flushing by injecting disk writes between consecutive disk reads. However, a single buffer page is still required to hold the result of each disk read and the read operations can still exhibit poor performance if the input workload has poor data locality. In contrast, BOSC's low-latency logging technique can accommodate a much larger number of disk writes, its use of sequential disk I/O to commit pending updates greatly improves the sustained disk update throughput and it does not rely on NVRAM to ensure data durability.

There has been a long line of research on efficient file system metadata update techniques that ensure metadata consistency with minimal performance overhead. HyLog [77] further reduces the performance overhead associated with LFS's cleaning [78], by treating hot and cold pages separately. The soft update technique [79, 80] avoids synchronous metadata writes by exploiting dependencies among metadata updates and makes it possible to aggregate updates as much as possible to improve the disk I/O efficiency. One problem with soft updates is that it is metadata-specific and thus needs to be tailored to each type of file system. Also, the above metadata update techniques focused mainly on the latency but not the throughput of metadata updates.

Efficient file system metadata update techniques that ensure metadata consistency with minimal performance overhead have received significant attention in the last two decades. *WAL (Write-Ahead Logging)* [81, 82] and shadow paging [77, 78, 83–85] group related metadata updates and commit them atomically to ensure metadata consistency. Performance benefits of *WAL* mainly come from sequential disk writes and group commit.

Much work [86–90] has been done to optimize the disk I/O performance for inserting and querying index data structures. One particularly interesting line of research in this area is the cache-oblivious data structures and algorithms [91–94]. Take a binary tree B of height H for example. This tree is abstracted into a 2-level abstract tree AB, whose root corresponds to the first $\frac{H}{2}$ levels of B, and each of whose leaf nodes corresponds to a $\frac{H}{2}$-level subtree of B. Each node in AB is then recursively abstracted in the same way until the size of each final abstract tree node is smaller than a pre-defined threshold T. This linearization strategy for tree data structures, known as the van Emde Boas scheme, substantially reduces the number of disk accesses required in the tree look-up process if T is smaller than the cache line (page) size. The performance improvement of cache-oblivious data structures mainly comes from

the fact that they put portions of a tree that are likely to be accessed together during the look-up process in the same units which are transferred in the memory hierarchy. With this set-up, when a transfer unit is fetched into the main memory, it is expected to service multiple accesses to the unit before it is evicted.

There have been several research efforts on the bulk update problem, which attempts to speed up index updates in the presence of a continuous stream of inputs to a database, which require real-time updates to its indexes. Arge et al. [95, 96] proposes a bulk update mechanism for dynamic R-trees, whereas Procopiuc et al. [97] describes a scalable bulk update algorithm for kd-trees. The basic idea behind these schemes is to hold the inserted input records in the internal nodes as long as possible and copy them sequentially to grow the tree when the internal nodes are filled up. In the *buffer tree technique* [91], incoming updates to a B^+ tree are written to the smallest B^+ tree that can fit into the main memory. Merging is implemented as a background operation to take advantage of large sequential writes. However, read query performance again is sacrificed because multiple B^+ trees have to be queried before the final result can be computed. Graefe [98] proposes a novel technique to improve the de-fragmentation and reorganization performance of B^+ tree. The idea is to use a logical pointer called *fence* instead of a physical pointer to sibling B^+ tree leaf nodes, to limit the performance overhead of migrating B^+ tree leaf nodes. However, this scheme optimizes the performance of insert operations but not update in-place operations, because the latter needs to fetch target leaf nodes before modifying them.

HDFS [3] uses append-only writing to mitigate random writes, but that comes at the expense of low locality in reads. Its optimized for batch processing systems like MapReduce [14]. HBASE [13] is built over HDFS to improve upon real-time read/write accesses.

BOSC is different from these database-index optimization schemes in three ways. First, BOSC is application-independent and requires only minor modifications to the database indexes built on top of it. Second, BOSC speeds up the disk access performance through request batching and sequential commit, without requiring any additional data structure copying. Third, BOSC can handle arbitrary index modifications, i.e., insert, delete and in-place update, but most bulk update schemes are optimized for streaming inserts.

2.5 Fast Disk Logging

Applications that do intensive data write operations often bottleneck on slow I/O bandwidth. A typical solution is to do delayed writing like Aries [105],

which involves logging followed by an asynchronous write. The bottleneck now shifts to the logging operation and if the logging record size is small, the underlying storage has to manage high throughput with low latency even in cases of small random logging updates. Much research has been done on improving the logging interface, like the append-only logging technique [106], and in this section of the survey we will discuss a few important research works that highlight the core issues in an efficient logging system.

The idea of writing data to disk at the position where disk head happens to be, can be traced back to as late as Trail [107]. Though Trail aims at the problem of minimizing seek delay and rotational latency, it's not trivial to implement it these days. It involves having accurate control over disk geometry details like rotational latency, seek latency, number of sectors in each track, zone coding, bad sectors mapping and other finer details. It's much tougher to implement this idea these days because of the advanced disk compaction techniques and more importantly disk manufacturers no longer supply the inner details of disk layout due to complicated disk management techniques and also due to competitive market. Multiple prior research efforts [108–111], similar to trail have been proposed, that target specific workloads using accurate disk geometry predictions. Lumb et al. [112] propose the idea of setting NCQ length to 2 and then utilize the disk seek and rotational latency to do some useful background work. *Beluga* also uses limited command queueing technique but also builds a sophisticated pipeline exploiting disk subsystem to the fullest extent. Yet another strikingly differentiating feature is in the added burden of these Trail like approaches, to maintain a map of used and free blocks on disk, in order to place the incoming data accurately on an unoccupied block, and at the same time avoid track switch delay. *Beluga* avoids these by sweeping through the disk sequentially, without leaving behind any holes in the process. As a result, *Beluga* doesn't need to maintain any mapping information of the used and freed blocks.

Gallagher et al. [113] propose to skip N number of blocks depending on the observed latencies at each portion of the hard disk and hence makes it extremely hardware dependent. Moreover, adopting this technique on modern disks with advanced NCQ capabilities is very time consuming. More importantly, they propose a model to avoid disk rotational latency by idling during the time the disk head skips the requested number of blocks. Our approach totally eliminates any sort of latency and achieves the best possible theoretical latency because the disk head never moves without doing any useful work.

The complexity of modern disk drives as elucidated by Gim et al. [114], an in-depth explanation of the Linux kernel storage subsystem in the book [115] gave us a good understanding of the complex sector layout schemes and the

difficulties associated with the accurate estimation of the modern day hard disk geometry.

Logging disk Array [116] uses the RAID technology to handle small writes problem and NVRAM buffer to provide persistency to the cache. The buffer is flushed periodically to disk(Raid-5) when sufficient data is built up. Since RAID uses stripe size as the basic unit of data transfer to disk, NVRAM buffer is structured to hold data in multiples of the stripe size. This idea helps aggregate smaller writes and then write it at one shot to disk in units of stripe size so that no additional overhead is incurred in the transfer process. Though latency in writing to NVRAM buffer is very low(in order of microseconds), flushing NVRAM buffer to disk is not a trivial task. Though optimal size is chosen in units of stripe size, there are various other factors which determine whether the disk is utilized to the best extent. That's where *Beluga* intends to break down the performance metrics and show how tuning certain parameters can help achieve best results. Another important factor to note is that NVRAM is a costly hardware resource, which can be avoided if the inexpensive SATA disks can be carefully tuned to yield same or even better results. In many situations, writing to NVRAM can yield very slow response times [117].

Log structured file system(LFS) [118] is another major solution to handle small buffer size writes. The entire file system is organized as a sequential log, which converts writes from user application as append to the underlying log structure in the File System. But logging operations require persistent write to disk and hence synchronous writes are required, which obviously yields a very low performance on a naively implemented LFS. Advanced LFS techniques like [119–124] use NVRAM or flash to make LFS handle synchronous writes efficiently, but both NVRAM and flash are costly hardware alternatives. Though flash based disks provide very high throughput and very low latency, erase cycles are very slow and hence flash disks' performance goes down when its utilization factor goes up. Also, the basic block size of flash ranges from kilobytes to megabytes and is much higher than the sector size of typical magnetic hard disks. The erase operation in flash devices requires the block size to be of bigger size to get optimal results. However, having a bigger block size increases the latency of smaller requests, which need to be aggregated to form a bigger block size. Flash logging [125] technique uses an array of USB flash devices to provide a fast logging infrastructure. The work proposes to use commodity USB devices as an alternative to expensive SSD based logging systems. The author discards modern day magnetic disks as ill suited for small sequential writes based on a naive logging implementation on SAS disks. *Beluga*'s evaluations convincingly show how commodity hard

drives can be used to extract comparable performance as that of the expensive flash based devices.

Phase Change Memory(PCM) [126] is a faster alternative to flash based disks but because of its smaller density and higher cost, it's not easy to be adopted in near future. Mohan et al. [127] propose to use PCM as the first choice for logging, since the speed of PCM is up to four orders of magnitude faster than that of flash based disks [126, 128], thereby guaranteeing very high throughputs and very low latencies.

Dynamo [129] and Cassandra [130] performs in-memory logging and the logging system is spread across multiple systems so that even if one system crashes data can be recovered from other machines. Various techniques are used to isolate catastrophic failures to ensure high reliability. With the increasing technological advances in network speed, data can be transferred across systems in a very short time thereby providing low latencies. However this comes at the expense of expensive RAMs and high end networking hardware. Azure [20] and HDFS [3] uses journalling and append-only logging to maintain data persistency.

Dual actuator [131] proposes to reduce synchronous write seek time using an accurate disk head prediction technique, It uses an additional hardware actuator and a set of disk heads to service read I/O requests. While one disk head actuator is dedicated for servicing disk write I/Os, another disk head actuator is dedicated for servicing read disk I/Os. However the author makes an assumption that disk head prediction techniques can be easily adopted, but unfortunately it is no longer easy with modern disk drives. Additionally, this technique requires additional hardware and hence is not applicable to existing storage devices. However, *Sungem* is able to achieve near zero seek times, without any additional hardware alteration and without the need to predict the disk head position.

2.6 QoS for Distributed Storage Systems

QoS aware storage virtualization techniques have been an active field of research for the past several years. The evolving nature of the storage systems from directly attached storage systems to clustered storage systems have opened numerous challenges in enforcing QoS guarantees. In this section of the survey, we highlight a few important research works that address the QoS challenges in a distributed storage system, and we compare them to our proposed QoS model of *Cheetah*.

2.6.1 Description of QoS specification in Service Level Agreements

QoS specifications can be described in a service level agreement (SLA) in several different ways, depending on the requirements of the application and the functionalities provided by the storage system. The requirements of an application is in terms of latency, throughput, availability and other similar attributes. Wilkes [133] gives an exhaustive overview of different types of QoS specifications on traditional storage systems and hence we do not repeat them in this survey. With a distributed storage model, availability and replication factors play a major role in controlling the overall performance of the storage system. Therefore, QoS specifications are getting more complicated, by including advanced and abstract requirements like consistency. Pileus [134] gives a good overview of different consistency-based service level agreements for a cloud-storage system. Though these QoS specifications give a tight control over the performance of the storage system, tenants hardly get it right. The QoS specifications are all based on the characteristics of the physical hardware, like "minimum bandwidth of 1000 IOPS, with average I/O request size of 8KB", etc. But we believe that tenants would benefit the best if the QoS specifications are all based on just the characteristics in the tenant's applications, like "minimum processing rate of 1000 objects/second, where an object could be of any size". The storage system should ideally be able to automatically convert application-level QoS requirements into system-level QoS requirements with a high level of accuracy. Stonehenge [135–137] provides QoS guarantees at virtual disk granularity and the underlying hardware is a set of directly attached disk arrays. QoS guarantees are made on multiple dimensions, availability, bandwidth, capacity, delay and elasticity, where all these attributes are derived from the application-level requirements. We built *Cheetah* to extend the QoS specification model in Stonehenge, into a more generalized and distributed storage framework of DISCO.

2.6.2 Granularity of QoS Enforcements

In a virtualized system, QoS guarantees can be done at different levels of granularity. At VD-level of granularity [136], each VD is guaranteed with a performance objective. At VM-level of granularity [138, 139], each VM is guaranteed with a performance objective and all the hardware resources allocated to the VM collaboratively share the VM's QoS reservations. At VDC-level of granularity [140], a group of VMs and VDs that belong to a single application, will need to collaboratively share the physical hardware resources and hence the QoS guarantees have to be decomposed accordingly to each VM and VD in

the VDC. Gulati et al. [140] define a software system called software resource pool (SRP) that groups related VMs and provides QoS guarantees both at VM and SRP level of granularity using hierarchical resource allocation. QoS guarantees are made to ensure minimum and maximum throughput, and proportional shares to prioritize VMs when capacity is constrained. QoS settings are decomposed to all the VMs in proportion to their shares and it done using a resource pool tree that helps provision resources at run time in a very efficient manner. Depending on the average latency observed for disk access requests on a host, length of the disk queue is controlled and hence the storage nodes are ensured to be not overloaded. VMware's storage DRS [141] provides QoS guarantees at data store cluster granularity, where data store cluster is a group of disk arrays called data stores. Tenants specify space and latency requirements for the entire cluster

2.6.3 Location of Collecting Statistics

In order to enforce the QoS specifications, it is necessary to measure accurate statistics of the input workload and the performance of the storage devices. In a small-scale storage system, it is relatively straightforward to measure the input workload information by directly measuring it on the source nodes(which are typically VMs or hypervisors corresponding to the given VMs). However, in a cloud-scale clustered storage system, VDs are spread across physically different machines, and hence it is a challenging task to co-ordinate between the source nodes at real-time.

Action	**Black-box**	**White-box**
Location of stats collection	Source Node	Storage Node
Accuracy of stats collection	Estimation	Exact
Adaptation to hardware changes	No effect	Complicated
Flexibility in QoS Management	Generic view of storage	Extract specifics of hardware

Table 2.2: *Comparison of white-box vs black-box stats collection techniques in a cloud-storage system*

In order to extract the performance statistics from each storage device, there are two well-known techniques, namely, white-box and black-box, whose relative advantages and disadvantages are compared in table 2.2. In black-box technique, the storage nodes do not give any detailed information about its structure or the nature of devices within the storage node. A storage node could be a disk array holding 5 magnetic disks, 10 SSD disks, the disks might

be of different size, speed, and the disks might be interconnected with either PCI cards, ethernet or fiber optics. A white-box technique [142] allows the QoS system to collect performance measurements from within the storage nodes, which ensures in capturing highly accurate load information. This ensures more visibility of what's inside each storage node, which could consist of multiple DAs, and thus could leverage this visibility to distinguish between volumes with high access locality and volumes with low access locality and treat them differently. On the other hand, in a black-box technique, the performance numbers have to be collected from the source nodes using end-to-end I/O latency of every I/O request. PARDA [143] proposes a black-box based estimation technique that can isolate the network I/O latency factor from the observed I/O latency, and helps estimate the input workload locality through the measured I/O latency. Though, a black-box technique loses on accuracy, its advantageous from the aspect of adapting to hardware changes, because the black-box technique anyway doesn't know about what's behind a storage node. A white-box technique has to understand the changes in hardware structure and extract useful information before the modified storage node starts accepting I/O requests. However, it is not a strict requirement because a white-box technique can tradeoff on the amount of dependency on hardware devices and might even be designed to be invariant to the hardware changes. *Cheetah* is designed on this model because it is an important requirement, especially in a cloud-storage environment involving large variations in physical hardware resources. Both these approaches claim to be flexible in managing the QoS management policies in their own right. While a black-box technique claims to be flexible because of its generic view of storage nodes, especially making the best use of a NAS/SAN environment, a white-box technique claims to be flexible because it can make informed decisions on load balancing, flow control and other important QoS functionalities using the specific details and exclusive control over storage nodes modeled on a JBOBOD architecture.

A challenging task in both the black-box and white-box techniques is in a clustered storage system, when the workload on a storage node is submitted from different source nodes. In order to make better routing decisions, it is necessary to combine the workload locality information from the source nodes and load information on storage nodes, while minimizing the network bandwidth consumption. While, it is challenging for a black-box technique to gather load information of a storage node that is shared by multiple source nodes, it is challenging for a white-box technique to gather workload information from the source nodes. BASIL [144] uses black-box technique and suggests to aggregate the load information collected on different VDs, by communicating with all such source nodes spread across several physical systems. However, such a

technique will lead to increased network bandwidth consumption and BASIL doesn't provide convincing results to justify such an aggregation policy. In *Cheetah*, we use a white-box technique and collect all measurements on the storage nodes only. A central controller is used to periodically aggregate the input workload information of the VDs and the load information of the DAs to determine ideal load balancing weight distribution. Since a cloud storage system is expected to have more number of source nodes than the storage nodes, *Cheetah* measures accurate performance numbers and also minimizes the network bandwidth consumption to a great extent, when compared to a black-box technique.

2.6.4 Performance Isolation

An important feature of all storage QoS systems is in ensuring performance isolation. Argon [145] and Fahrrad [146] uses time sliced allocation of disk accesses to avoid interference between two workloads to a great extent. Facade [147] uses a combination of earliest deadline first (EDF) [148] disk scheduling algorithm and an ad-hoc disk queue size manipulation technique to control the disk utilization efficiency. Pisces [142] provides performance isolation by using a combination of few novel techniques. First, it distributes a tenant's share across storage nodes, using a weighted allocation algorithm that considers both the workload locality and the load on storage nodes. Second, it uses a variant of weighted fair queuing [149] algorithm within each storage node to ensure performance isolation and fairness between the workloads that share the same storage node. Virtual Clock (VC) based schedulers like CVC [136] and CFVC [135] ensure better control over raw disk bandwidth utilization because of the intelligent techniques to identify and distribute slack in inactive workloads to active demanding workloads. *Cheetah* adopts CFVC scheduler in each DA to ensure maximum raw disk bandwidth utilization while ensuring performance isolation between different workloads on the DA. Though, *Cheetah*'s higher level ideology is similar to that of Pisces [142] in ensuring fairness and performance isolation for all the I/O workloads, *Cheetah* differs in the way the load balancing is done and also differs in the QoS aware disk I/O scheduler used within a storage node. Throttling-based mechanisms [150] rely on delivering consistent I/O bandwidth for each I/O workload by using feedback-based mechanisms that throttles the data flow rate to control the latency of each I/O request in the data workload. By maintaining a consistent disk I/O throughput performance isolation is guaranteed, but it is quite a challenge to accurately control the latency of an I/O request in a large-scale distributed storage system. Variations of token-bucket approaches [151–153] propose to throttle the bandwidth and bursts in the incoming workload by assigning fixed

quotas or tokens to each workload. By strictly tokenizing different workloads that share a storage resource, performance isolation is guaranteed at all times but it comes at the cost of under utilization of hardware resources.

2.6.5 Provisioning Hardware Resources

Provisioning of hardware resources is a critical task in a virtualized storage system because it is extremely challenging to strike the right balance between cost factor to provision additional hardware resources and the accuracy with which QoS guarantees are enforced. Minerva [154] models the requirements of an automated storage system as a detailed input specification, indicating performance requirements, workload statistics and storage device capabilities, as a constraint-based optimization problem, to provision the hardware resources automatically. Hippodrome [155] iterates multiple times over the process of characterizing the workload and creating a new provisioning solution until all the workload requirements are handled. BASIL [144] uses IO latency as the primary metric to characterize both the input workload and storage nodes, and provides optimal strategies to place the VDs on appropriate storage nodes. Stonehenge [137] uses 2 metrics: Pusage and Pservice to handle admission control management. Pusage metric measures the aggregate disk bandwidth used and Pservice metric measures the ratio of actual to worst case delay expected. Based on their observations, the pservice metric is shown to be directly proportional to the number of virtual disks on the server and hence for a new input workload, Stonehenge does't profile the new application for a long time to understand it's disk IO patterns, rather it uses the pservice metric to estimate the hardware resource requirements and automatically provisions them accordingly.

2.6.6 Load Balancing

I/O workloads exhibit different kinds of fluctuations in the workload locality. A short term fluctuation is a small period(less than a second) of bursts in the workload and is usually absorbed either by the software disk schedulers located on the DA or by the hardware disk schedulers used by the disk controllers. A long term fluctuation is an overall change in the workload locality pattern as a result of some permanent changes in the workload, and such fluctuations typically last for several hours before returning to the expected locality pattern or can even be a permanent deviation from the expected workload locality. Therefore, in the event of such long-term fluctuations in the workload, either the tenant reconfigures the QoS specifications or the system automatically

adds or removes hardware resources, such that the QoS specifications are enforced accurately. However, there are mid term fluctuations too, that exhibit variations in the workload locality in the order of few minutes, and it is these type of workloads that pose serious challenges in enforcing the QoS guarantees because it is not sure whether to reconfigure the system settings or to wait until the fluctuations disappear.

In VMware's storage DRS [141], when a VM is seen to overload its storage node, the VM is isolated and migrated to a least loaded storage node within its cluster, where a cluster is a hard partition of a set of storage nodes. In BASIL [144], when a storage node is overloaded, VDs are migrated to different storage nodes using a pair-wise assignment algorithm. By pairing VDs of minimum and maximum loads, the pair of VDs are expected to balance the overall load in the system. Its a common practice to migrate VDs from overloaded DAs to new or least loaded DAs [155–157], but the cost of data migration can be prohibitively long. Therefore, BASIL suggests to use Storage VMotion to show that data migration from one store to another store can be automated and need not block the VMs. They use workload characterization to understand the load pattern to manage the data transfer. HP's AutoRaid [157] maintains a hierarchical storage system that has RAID1 setup on the higher layer to ensure higher throughput and lower latency and in the lower layer of the storage system, it has a RAID5 setup to provide additional redundancy. The data migration between RAID1 and RAID5 layers are done automatically in the background, transparent to the user applications.

A disadvantage with the data migration technique is the need for migrating the entire VD to a different storage node. The cost of migration can prohibitively bottleneck any I/O accesses to the affected storage nodes and may not benefit for workloads with frequent short-term fluctuations. Hence the load balancing feature through data migration is more of a reaction to an imbalanced load rather than a proactive measure like ours where best effort is made to balance the loads uniformly across all the DAs.

In Azure [20], storage system consists of multiple clusters of storage nodes and load balancing is done only within a cluster of storage nodes. Each read request is associated with a strict deadline timestamp and if the deadline cannot be met, the storage cluster returns the query to the source node that submitted the I/O request. The source node increases the deadline timestamp and again submits the data until the data is successfully processed by the storage cluster. Since Azure uses erasure coding to replicate data for high availability, a data object is striped and replicated across several storage nodes. To handle a read I/O request, it has an option to either read all the various data fragments corresponding to the data object from a set of storage nodes or

to reconstruct the data object based on analytical methods, from a different set of storage nodes. The decision to reconstruct or not, is made dynamically depending on the real-time load on the storage nodes.

Zoolander [158] replicates data for providing high availability and to ensure predictable performance guarantees. When there is contention for data access, which is reflected in the slow I/O latency of the data workload, the entire storage node is replicated to distribute the load and bring down the I/O latency. Though it enforces performance guarantee, it comes at the cost of increased hardware utilization.

Pisces [142] proposes an unique approach to load balancing the storage nodes using reciprocal swaps. The crux of the algorithm is the give-and-take policy: if a tenant t takes some share of the storage node N from a tenant u on N, t must give an equivalent share back to u on another storage node M. This might not be feasible on a large-scale storage system where t and u might not share more than one storage nodes at all, and its an additional constraint to the admission control algorithm. Even otherwise, if you take some share from one tenant and give it to another tenant, the reciprocal swap procedure has to continue until there is an equilibrium in the entire storage system, which might necessitate reversing some actions. In order to avoid such a scenario, *Cheetah* uses a multiple iterative procedure to minimize the possibility of overloading a storage node.

2.6.7 Extending QoS Ideas From Non-Storage Systems

Paragon [159] uses collaborative filtering techniques to identify how well an incoming application will run on different types of platforms. Rather than profiling the applications to understand their workload behavior, some key components of an application are matched with a database consisting of offline profiled applications that are run on different types of platforms. Storage virtualization introduces totally different kind of requirements than server virtualization, and the idea of adopting robust analytical methods to characterize the input workload rather than profiling it at run time, is difficult to extend to storage virtualization because of too many unpredictable variables in estimating the behavior of an I/O request on a storage device.

QoS management in networking systems has been studied for a long time and though they cannot be adopted straight away on storage systems, the fundamental ideas can be borrowed as there are some similarities between networking and storage domains. For example, to mitigate congestion in a high speed switch, it is a common technique to build a crossbar to connect input ports to output ports and then transfer the cells in input queue through the connection established between input and output ports. In order to quickly es-

tablish a pair of input and output port such that the input port has atleast one queued cell to transmit, parallel iterative matching (PIM) [160] technique uses multiple iterations to pair an unmatched input with an unmatched output. To keep the number of iterations down and to identify the pairs at run time with very low latency, randomness and parallelism is used to good effect. Though the random mapping technique in PIM helps reduce the number of iterations, it is relatively time consuming and more importantly it can cause unfairness leading to large average latency in the network. iSLIP [161] algorithm replaces the random mapping with weighted round robin (WRR) [162] technique and shows that without compromising on the overall performance, overall fairness is ensured. The fundamental technique underlying PIM and iSLIP is to divide the input-output mapping process into multiple iterations, where each iteration maps one input to one output. This technique is adopted into *Cheetah*'s load balancing algorithm and is explained in detail in Section 7.4. Gopalan et al. [163] leverage the networking concepts of managing run-time packet delay distribution into storage domain and support long term storage bandwidth guarantee for tenants.

Chapter 3

Scalable Deduplication and Garbage Collection

3.1 Introduction

To ensure data protection in the SDDS system, DISCO backs-up the data from VDs to a safe archival storage. The data backup operation is expensive because it incurs additional disk storage overhead in the archival storage and consumes a high number of CPU cycles to manage the backup operation. Therefore, the SDDS system allows tenants to configure the periodicity of the backup operation, for each VD the tenant uses for his storage. As explained in section 1.2.3, DSS performs several incremental backups and full-backups, for every VD in the SDDS system, depending on how a tenant configures to handle the backup operation for each of his VDs. Since the number of duplicates in a full-backup operation are pretty high, the challenges in deduplicating a full-backup data is not as challenging as to deduplicate data in an incremental backup. Therefore in this work, we focus primarily on the incremental backup.

Due to increased data sharing in the backups [164], data deduplication has become a key functionality requirement [32, 34, 37] in a backup operation. The two key metrics for assessing the performance of a data deduplication technology are *data deduplication ratio or duplicity*, expressed as the percentage of duplicate blocks in the backup VD before deduplication, and *data deduplication throughput*, expressed as the number of backup data blocks that a data deduplication engine processes per second, to determine whether it is a duplicate or not.

To find duplicates, it is simply impractical to do a byte-by-byte comparison with all the blocks in the backup archive. Hence a well known technique of fingerprinting [31] is used, where a block's contents are cryptographically

hashed into a fingerprint of typically 20 bytes size [35], and the fingerprints are then used for duplicate comparisons. In the DISCO setup, DSS initiates the incremental backup operation for each VD at pre-configured periodic intervals. A backup client which is a part of DMS, employs *block change tracking* (BCT) [165] technology to keep track of all changes to a VD in a delta list, and at backup time, transmits the delta list to the backup server which is a part of DSS. For each modified disk block in the delta list, the backup server fetches the corresponding fingerprint values from their corresponding SNs and creates a stream of fingerprints to be passed on to the deduplication engine. The deduplication engine which is a component of the DSS could either be located on the same physical machine as the backup server or could be located on a dedicated standalone system. For each fingerprint in the fingerprint stream, the data deduplication engine replies to the backup server if it is a duplicate of some existing fingerprint in its database or not. Though the chances of fingerprint collisions are rare and known to be lesser than the probability of a hard disk failure [31], there is still an element of risk where a valid user data could be freed, leading to data corruption and data loss. In order to eliminate the risk factor, for all the identified duplicate fingerprints, a separate background thread fetches the real data blocks corresponding to the identified duplicates from the corresponding SNs, and it does a byte-by-byte comparison to confirm the duplicates. The confirmed duplicate blocks are then garbage collected and released to the available storage space.

For every disk block stored in the repository, the deduplication engine needs to organize the fingerprints into an index structure to determine whether an incoming disk block is a duplicate or not. So fundamentally the data deduplication problem is one of index look-up, where the index is too large to fit into memory. Worse yet, there is no spatial locality among neighboring fingerprint values because fingerprints are generated by cryptographic hash functions. As a result, in the worst case, every look-up into the fingerprint index could result in at least one disk I/O, which drastically slows down the data backup process.

Several solutions to the performance problem associated with fingerprint index look-up have been proposed and they are all based on the following assumptions:

- Data being shared among users, e.g., a PowerPoint file or a photo image file, is typically larger than a single disk block.
- A data sharing unit typically spans a consecutive sequence of logical blocks in a VD.

Taken together, they suggest that when multiple instances of a data sharing unit are backed up, multiple identical sequences of logical blocks are likely to appear in the backup streams. There are two ways to exploit the fact that identical sequences of logical disk blocks are repeated multiple times. First, the fingerprints of the blocks in a recurring sequence are stored in a container, and the entire container is fetched into memory whenever any fingerprint of an incoming block in a backup stream matches a fingerprint in the recurring sequence. This technique, used in Data Domain [37], essentially corresponds to *prefetching*, because fingerprints associated with a basic data sharing unit are likely to be accessed together, and thus should be brought into memory via one disk I/O operation to amortize the disk access overhead. Second, instead of storing every fingerprint in the fingerprint index, it is sufficient to put into the fingerprint index only one of the fingerprint values associated with a data sharing unit, because it takes only one fingerprint match to bring in the remaining fingerprints in the same data sharing unit. This technique, known as *sparse indexing* [6], corresponds to *sampling* of the fingerprint index, and is able to significantly reduce the fingerprint index size and thus increase the probability of looking up the fingerprint index without incurring disk I/O.

We built a data deduplication and garbage collection engine called *Sungem* [166] that is designed to remove duplicate blocks in incremental data backup streams. *Sungem* features three novel techniques to maximize the deduplication throughput without compromising the deduplication ratio. First, *Sungem* puts *related* fingerprint sequences, rather than fingerprints from the same backup stream, into the same container in order to increase the fingerprint prefetching efficiency. Second, to make the most of the memory space reserved for storing fingerprints, *Sungem* varies the sampling rates for fingerprint sequences based on their stability. Third, *Sungem* combines reference count and expiration time in a unique way to arrive at the first known incremental garbage collection algorithm whose bookkeeping overhead is proportional to the size of a VD's incremental backup snapshot rather than its full backup snapshot. We evaluated the *Sungem* prototype using a real-world data backup trace, and showed that the average throughput of *Sungem* is more than 200,000 fingerprint lookups per second on a standard X86 server, including the garbage collection cost.

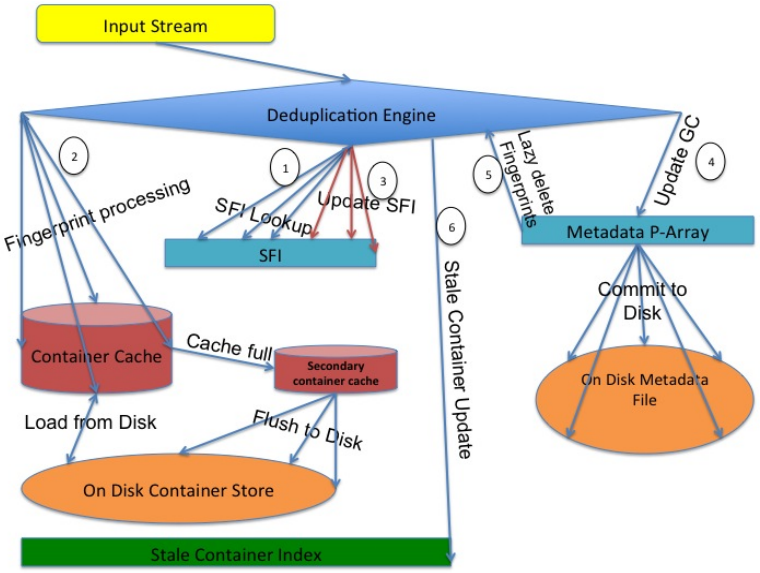

Figure 3.1: *Abstract view of the proposed data deduplication engine*

3.2 Our Approach

3.2.1 System Architecture

Figure 3.1 shows the high-level data flow of the incremental data backup system in DSS, that uses *Sungem* as the deduplication engine. To backup multiple VDs, fingerprint stream for each of them is prepared individually. Upon preparing a fingerprint stream, the DSS partitions it into fixed-size *input segments*, where each segment corresponds to a sequence of blocks with *contiguous* logical block numbers. For each input segment, *Sungem* first looks it up in the *sampled fingerprint index* (SFI), which contains sampled fingerprints from previously seen fingerprint segments and the IDs of the on-disk containers to which these sampled fingerprints belong. For every fingerprint in an input segment that hits in the SFI, *Sungem* brings its corresponding container into memory to determine if the input segment indeed contains any duplicate instances of any previously seen fingerprint sequence. To speed up container fetching, *Sungem* uses a memory-resident *container cache* to hold containers recently fetched from the on-disk container store, which are organized in an LRU order. For every input fingerprint, *Sungem* either declares it as a duplicate and returns the physical block number of the physical block in the repository that already has the same fingerprint, or declares it is not a duplicate. DSS uses the outputs from *Sungem* to create the underlying representation for a backup snapshot.

3.2.2 Fingerprint Segmentation and Placement

Intuitively, a *Sungem* segment is meant to capture the notion of a data sharing unit. From an input fingerprint stream, *Sungem* first partitions it into multiple fixed-size *input segments*, where each segment corresponds to a sequence of fingerprints whose associated blocks have consecutive logical block numbers. Each input segment formed this way could contain zero, one, or multiple *stored segments*, which are segments that were previously seen and already stored in the repository. To identify the stored segments contained in an input segment, *Sungem* queries the SFI with every fingerprint in the input segment. If a SFI query results in a hit, the SFI returns one or multiple $\langle containerID, segmentID \rangle$ pairs, each of which corresponds to a potential stored segment that contains the query fingerprint.

After going through every fingerprint in an input segment, *Sungem* brings into memory all the containers indicated in the responses of the SFI hits. Each container contains a *per-container fingerprint index* that maps a fingerprint into an offset inside the container that holds information about the fingerprint's associated disk block, e.g., its physical block number. In addition, each container contains a *segment index*, which maps a fingerprint's offset inside a container to all the stored segments in which the fingerprint participates. *Sungem* keeps containers in an on-disk container store, and uses an in-memory container cache to keep containers that were recently fetched from the disk so as to reduce the disk access cost associated with fetching containers.

For each hit fingerprint, *Sungem* fetches the associated container from disk, consults with the container's fingerprint index to obtain the offset location of the hit fingerprint, uses this offset information to identify all stored segments in this container that include the hit fingerprint, and finally performs fingerprint-by-fingerprint comparison between each such stored segment and the input segment, anchored at the hit fingerprint. After this process, each fingerprint in the input segment either matches some fingerprint in some stored fingerprint, or does not match any previously seen fingerprint. Each maximal-length sequence of fingerprints that matches some stored fingerprint form a *matched subsegment*, which could be identical to, a subset of or a superset of an existing stored segment. Every matched subsegment that is not identical to any existing stored segment forms a new stored segment, and *Sungem* places it in the same container as the *first* fingerprint of this subsegment. Note that a new stored segment derived from a matched subsegment could match partially multiple existent stored segments, but *Sungem* places it in the same container as the first matched stored segment. When a new stored segment is a superset of an existing stored segment, *Sungem* is putting related stored segments in the same container. The fingerprints in the input segment that do not match

any stored fingerprint form a new stored segment, and *Sungem* places it in the default container, which holds newly discovered stored segments from an input backup stream that are not related to any existing stored segments. If none of the fingerprints in an input segment hits in the SFI, the entire input segment forms a new stored segment and is stored in the default container. As an input backup stream's default container grows in size and becomes X% full, where X is the *fill-up threshold*, *Sungem* allocates and switches to a brand new default container, so that the old default container can hold future stored segments that share common fingerprints with its stored segments.

In previous designs [6, 37], stored segments are assigned to the same container because they appear temporally close to each other. In *Sungem*, stored segments are assigned to the same container either because they appear temporally close to each other or because they share common fingerprints. The rationale of putting fingerprint-sharing stored segments in the same container is to bring in as many potentially matching stored segments in one container access as possible. For example, a popular image may be used in three different PowerPoint slides, each of which in turn may be included in many PowerPoint files. *Sungem*'s segment-to-container assignment scheme enables the fingerprint sequences associated with the three PowerPoint slides to form stored segments that are put into the same container. Whenever a fingerprint corresponding to a block in the popular image appears in an input backup stream, *Sungem* could bring in the stored segments corresponding to the three PowerPoint slides by fetching a single container.

To avoid proliferation of stored segments, each stored fingerprint is allowed to participate in up to K stored segments. More precisely, for each fingerprint, *Sungem* only records the most recently appearing stored segments in which the fingerprint is in, because these segments are more likely to match future input segments.

3.2.3 Variable Fingerprint Sampling

When a new stored segment is formed, *Sungem* uses a fixed sampling rate to pick representative fingerprints from the stored segment and inserts them into the SFI. Ideally, the sampling rate should be high enough to capture most data sharing units, but low enough to keep SFI's memory space efficiently utilized. *Sungem* employs a variable fingerprint sampling scheme to achieve the best of both worlds.

Fingerprints sampled from the same stored segment and inserted into the SFI are treated as a single entity and called a *fingerprint group*. The last access time of a fingerprint group is the last time any of its fingerprints matches a fingerprint in a new input segment. The fingerprint groups in the SFI are

organized into a LRU list based on their last access time. When the SFI needs to evict fingerprints, it down-samples the fingerprint groups in the tail of the LRU list to 1 per group, and keeps on doing this until it exhausts all fingerprint groups above a certain age; after that it starts removing entire fingerprint groups to free up space.

When a fingerprint group's fingerprints have matched fingerprints in more than a certain number of new input segments since its formation, it becomes a *stable* fingerprint group, and the associated stored segment is considered a well-defined deduplication target. *Sungem* reduces the number of representative samples in a stable fingerprint group to one per group, because one fingerprint sample is sufficient to capture future instances of the associated stored segment.

The above two optimizations allows *Sungem* to apply a high sampling rate to new stored segments while effectively reclaiming memory space from older stored segments. If a stored segment later proves itself to be a useful deduplication target, *Sungem* reduces its sampling rate to one per segment. If a stored segment turns out to be a not-so-useful deduplication target, *Sungem* also reduces its sampling rate to one per segment.

3.3 Scalable Garbage Collection

In a data backup system like DISCO that supports data deduplication, a physical block may be referenced by multiple backup snapshots. Because a backup snapshot typically has a finite retention period, the number of references to a physical block varies over time. When a physical block is no longer referenced by any backup snapshot, it should be reclaimed and reused. There are two general approaches to identifying physical blocks in a data backup system that are no longer needed. The first approach is *global mark and sweep*, which freezes all active backup snapshot representations, scans each of them, marks those physical blocks that are referenced by these snapshots, and finally sweeps all the physical blocks in the entire storage system to garbage collect those physical blocks that are not marked in the mark phase. The second approach is *local metadata bookkeeping*, which maintains certain metadata for each physical block, and locally updates a physical block's metadata whenever it is referenced by a new backup snapshot or de-referenced by an expired backup snapshot. The first approach does not incur any run-time performance overhead but may require an extended pause time, which is proportional to the storage system size, and thus is not appropriate for petabyte-scale data backup systems. But, a number of commercial products have adopted modified mark and sweep approaches that minimize the pause time to a great extent, and

seems to be reasonably effective. However it is still batch-oriented rather than incremental as in the case of our algorithm. Consequently, *Sungem* takes the second approach, which incurs run-time performance overhead due to metadata bookkeeping. How to minimize this metadata bookkeeping overhead is an important design consideration of *Sungem*'s garbage collection (GC) algorithm. A detailed comparison of these approaches is described in Section 2.3.

DISCO maintains a backup snapshot table for every VD that it needs to backup. The backup snapshot table consists of several columns where the first column represents the logical addresses of the blocks in the VD, and the second column represents the corresponding physical addresses of the blocks in that VD's second snapshot, and so on. Since a large majority of the blocks remain unmodified between successive snapshots of a VD, the table is optimized to record entries only for those logical blocks that are modified in a particular snapshot with respect to the previous snapshot of the given VD. Effectively, we can assume that every incremental backup snapshot is represented by a logical-to-physical (L2P) map, which maps logical addresses in the incremental backup snapshot to their corresponding physical addresses. For a full backup snapshot, the L2P map contains mapping for all the active referenced blocks in the VD. In addition, the garbage collector maintains a *physical block array* (P-array) that maintains metadata for each physical block in the entire SDDS system.

3.3.1 Hybrid GC: Our Approach

The main weakness with the *reference count*-based [70–72] and *expiration time*-based [73] garbage collection scheme is that their performance overhead at backup time is proportional to the full size of the VD being backed up, rather than the size of the backup snapshot which corresponds to changes to the VD. The performance overhead is much smaller if incremental backups are taken. We propose a *hybrid* garbage collection algorithm, specifically targetted for incremental backup systems. It maintains both a reference count and an expiration time for each physical block, and its performance overhead at backup time is proportional to the size of an incremental backup snapshot rather than the snapshot's underlying VD.

An incremental backup snapshot consists of a set of entries each of which corresponds to a logical block that has been modified since the last backup. Each incremental backup snapshot entry thus consists of a logical block number (LBN), a before image physical block number (BPBN) that points to the physical block to which the logical block LBN was mapped in the last backup, and a current image physical block number (CPBN) that points to the physical block to which the logical block LBN is currently mapped. At backup

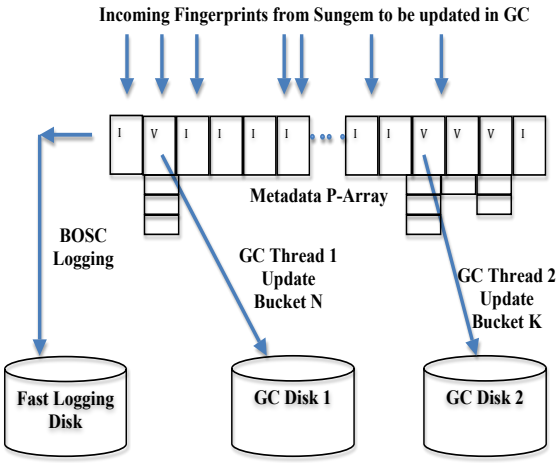

Figure 3.2: *Figure indicating metadata updates in garbage collection at backup time*

time, given an entry ⟨LBN, BPBN, CPBN⟩, the reference count of BPBN is decremented, the reference count of CPBN is incremented, and the expiration time of BPBN is set to the maximum of its current value and the current time plus the retention period of the VD being backed up. With this design, when the reference count of BPBN reaches zero, the physical block BPBN together with its expected expiration time is put into a recycle list. At garbage collection time, the physical blocks in the recycle list are scanned and those whose expiration time is less than the current time are garbage blocks.

In general, a VD has a current image and multiple backup snapshots. The reference count of a physical block in this algorithm keeps track of the number of current images, but not their associated backup snapshots, that currently point to it. The expiration time of a physical block records the time after which no backup snapshot will reference it. If there is at least one current image pointing to a physical block, this physical block cannot be a garbage block and its expiration time could be ignored. Whenever a logical block in a current image is modified, the current image no longer points to the physical block associated with the logical block before the modification, and the expiration time of this before-image physical block is updated to incorporate the retention time requirement of the current image's associated VD.

3.3.2 Batched Updates to P-Array

Hybrid GC algorithm implementation poses two major challenges. First, its accesses to the P-array, which is too large to fit into main memory, are largely

random and therefore could incur significant disk I/O overhead. Second, after a physical block is chosen to be recycled, the physical block's fingerprint needs to be removed from the rest of the deduplication engine, including the SFI, and the container holding the fingerprint. This fingerprint-removing overhead is a significant part of the garbage collection process regardless of the actual algorithm used to determine which physical blocks are recyclable.

To address the first problem, *Sungem* adopts the BOSC (Batched mOdifications with Sequential Commit) mechanism [74] (explained in detail in Chapter 5) to modify the on-disk P-array. More specifically, *Sungem* partitions the P-array into a set of chunks, and allocates a per-chunk queue for each such chunk. Each update to the P-array is put into the per-chunk queue associated with the P-array entry to be updated. Multiple background threads are used to sequentially scan the on-disk P-array, by fetching to memory each chunk whose per-chunk queue is non-empty, committing all updates in the chunk's per-chunk queue to the chunk, and writing the chunk back to disk. Using BOSC, *Sungem* requires mostly sequential disk accesses to update the P-array. Figure 3.2 gives an overview of the garbage collection setup.

To address the second problem, *Sungem* uses a lazy update approach to removing a recycled block's fingerprint from the deduplication engine. If the container holding the recycled block's fingerprint is memory-resident, *Sungem* deletes it immediately; otherwise *Sungem* marks the container as stale in a stale-container list, and queues the fingerprint to be deleted in the corresponding stale container entry in the stale-container list. When a container is brought into memory because of some HIT fingerprint processing, the stale-container list is first searched and if its found, the fingerprints in its queue are permanently deleted from that container. However, *Sungem* deletes the recycled block's fingerprint from the SFI immediately, because the SFI is always memory-resident. When a chunk of the P-array is brought in, the garbage collector scans the entries in the chunk to identify those whose reference count is zero and whose expiration time has expired, and puts them in the free list. This mechanism piggy-backs garbage collection with P-array accesses and thus reduces the garbage collection overhead to the minimum.

3.4 Parallelization Techniques for Deduplication and Garbage Collection

Historically, the size of a VD has increased from a few kilo bytes to several giga bytes over the last few years and continuing with this trend, the amount of data required to backup a VD is expected to increase, as well. However, the

duration of the backup process cannot increase at the same rate for obvious reasons. Hence the only way a backup operation can process large amounts of data without increasing the duration of the backup process is to increase the backup throughput using efficient parallelization techniques.

The sequential version of *Sungem* processes each input segment using the following steps:

1. Look up every fingerprint in the SFI,
2. Fetch into memory all containers referenced by all the hit responses from the SFI,
3. Perform fingerprint-by-fingerprint comparison between the input segment and all relevant stored segments in the fetched containers,
4. Identify new stored segments,
5. Put the new stored segments into proper containers,
6. Sample the new stored segment, and
7. Insert the sampled fingerprints into the SFI.

A simple strategy for K-way parallel data deduplication is to partition the fingerprint space into K portions, for example using a modulo K function, and assign each fingerprint in an input segment to one of the K nodes using the same partitioning function, where each node runs the sequential version of *Sungem*. This strategy is fully parallelized and relatively simple to implement, but it has a major flaw: the fingerprints associated with each data sharing unit are likely to form K stored segments. This means the total number of stored segments is increased by a factor of K, and more seriously the number of container-related disk I/Os required by processing of an input segment is also increased by a factor of K. Because the most likely bottleneck of a data deduplication engine is disk accesses associated with container fetches, a parallelization strategy that significantly increases the number of required disk I/Os is unacceptable.

3.4.1 Distributed Deduplication Algorithm Design

To overcome the limitation of the simple parallelization strategy discussed above, *Sungem* uses the following parallelization strategy, which requires a master node and K slave nodes. Given an input segment, the master node broadcasts it to all K slave nodes, each of which looks up every fingerprint *in*

its share (e.g. using some variants of modulo K partitioning function on the physical block address-space) in its local SFI, and returns to the master node those fingerprints that hit in its SFI and the associated hit responses. Due to the modulo K partitioning scheme, a slave node containing a SFI entry for a fingerprint need not hold the container storing that fingerprint. Hence the SFI entry is modified to hold additional information regarding the target slave node address where the container storing the given fingerprint is stored. Lets say the number of slave nodes that are addressed in the positive hit responses are $K2$, where $K2$ is independent of K. The master node again broadcasts the accumulated hit responses to the $K2$ slave nodes, each of which then fetches containers that are referenced in the hit responses and stored in its local disks, performs fingerprint-by-fingerprint comparison, and returns to the master node the hit/miss status of the input fingerprints it processes. The master node completes stored segment processing, and broadcasts to all the K slave nodes informing about the new stored segments and their associated containers. Each of the K slave nodes update their corresponding SFI accordingly.

With reference to the sequence of steps in the sequential version of *Sungem* described above, in *Sungem*'s parallelization strategy, all K slave nodes are involved in step 1 and 7, only $K2$ slave nodes are involved in steps 1, 2, 3, 5 and 7, and only the master node is involved in steps 4 and 6. Though the fingerprints are processed largely in a partitioned fashion, the processing at Steps 1, 2, 3, 5 and 7 are data-driven, i.e., whichever nodes hold the needed stored segment perform the associated computation. This strategy forms a single stored segment for each data sharing unit and stores it in only one of the $K2$ slave nodes. In order to scale-up the system to match higher demands in either deduplication throughput or the size of the storage system, this parallelization strategy could be further developed using minor changes to incorporate multiple master nodes. Additionally, this parallelization strategy doesn't increase the disk I/O activity compared to the sequential version, because it uses the same number of container-related disk I/Os for each input segment as the sequential version. However, the parallel version incurs additional inter-node communications cost, and may result in potential load imbalance.

3.4.2 Distributed GC Design

Similar to the K-way parallelization scheme mentioned above, the distributed GC partitions the P-array into equal-sized K portions using the same modulo function that the SFI uses to partition the physical address-space. In the last step of the K-way parallelization scheme, after each of the K slave nodes update their respective SFI, each of those slave nodes also update their local P-array. This parallelized GC operation is similar to the standalone GC al-

gorithm described previously except for the management of stale containers described in Section 3.3.2. When a fingerprint is marked to be recycled, since both the GC and SFI are partitioned on a common key (physical block number), the fingerprint is removed locally in that slave node's SFI However, the lazy approach scheme in the distributed GC can neither delete the fingerprint directly from the in-memory container structure nor queue the fingerprints in the stale-container list to lazily delete that fingerprint from its container, because the slave node holding the metadata of the given block need not essentially hold the container storing the fingerprint of that block. Therefore each slave node aggregates a list of fingerprints to be recycled and forwards such a list to the master node, and waits for acknowledgement from the master node. The master node broadcasts the list to K slave nodes and returns the acknowledgement that it receives from those slave node to the slave node that generated the list. When a slave node receives a request from the master node to delete a list of fingerprints, it can either delete the fingerprint from the in-memory container or queue the fingerprint in the stale-container list to lazily delete it at a later time when the container is brought into memory. However, this could cause a potential race condition which is better explained in the following example scenario. A stored fingerprint F1 is referenced as a duplicate by some incoming fingerprint F2, and immediate to that action the block corresponding to F1 is marked to be recycled on some other slave node. Since the slave nodes aren't synced, it is quite possible for the GC to recycle the block corresponding to F1 and then get a request to increment the reference count on that block because it was referenced by F2. This could lead to data corruption if the block corresponding to F2 is already recycled presuming F1 to serve as a duplicate to F2. To be theoretically correct in fixing such a race condition, the slave nodes could be synced for every GC operation, but its impractical, because the throughput of the entire deduplication process would then drop down drastically.

A simple alternate solution that the distributed GC adopts, is to maintain a timestamp in each stored fingerprint in a container, that indicates the time at which that stored fingerprint was last referenced as a duplicate. When a slave node receives a request from master node to delete the fingerprint, it would do so only if the corresponding container is present in memory, and the timestamp in the stored fingerprint is larger than a threshold T. Only if both these conditions satisfy, the fingerprint is deleted and recorded as "deleted" in the acknowledgement, otherwise the fingerprint is marked as "not deleted" in acknowledgement. The threshold T is chosen large enough to ensure that the fingerprint is successfully recycled in the GC, since it was last referenced as a duplicate to any other fingerprint. When a slave node receives the ac-

knowledgement from the master node indicating the status of whether the fingerprints in the recycle list are deleted or not, the slave node proceeds to recycle only those fingerprints that are successfully acknowledged as "deleted" and ignores the "not deleted" fingerprints. The ignored fingerprints if are not referenced as duplicates by any fingerprint, but are still marked as "not deleted" in the acknowledgement, will eventually be garbage collected whenever the container holding that fingerprint is fetched into memory.

3.5 Performance Evaluation

3.5.1 Evaluation Methodology

We have implemented *Sungem* in Java and successfully integrated it with DSS. To evaluate the effectiveness of the design decisions and the implementation efficiency of this prototype, we collected a real-world backup trace from a production environment, and used it in a trace-driven evaluation study.

Two evaluation metrics were used. The first metric is the data deduplication ratio, which is expressed as the percentage of duplicate fingerprints over the input fingerprints before deduplication. The second metric is the data deduplication throughput, which is measured in terms of the number of fingerprints that can be processed per second, including the overheads of both deduplication and garbage collection.

Trace Collection and Analysis

To derive a real-world trace of incremental data backup streams, we wrote a user-level tool that tracks and records changed files in a file system on a Windows machine within a period of time, and deployed this tool on 23 desktop machines of a research laboratory to collect the set of changed files every day on each machine for 10 weeks. Each of these 23 machines was used predominantly by a single user. We will refer to the resulting changed file trace as $userTrace$ in the following discussion. At the beginning of the trace collection period, the user-level tool traversed a file system, and for each file recorded into a database its last modify time and a 64-byte MD5 fingerprint for every 4KB block in it. At the end of every day, this tool traversed the file system, and compared the current modification time of each traversed file with its previously recorded modification time if it existed. If the previous modification time of a traversed file did not exist, the file was newly created. If the current and previous modification times of a traversed file were different, the file had been modified. In either case, the tool further computed a 64-byte MD5 fingerprint for every

File Type	Conjectured Append Behavior	Contribution Percentage (%)
VM-related Files	Append	35.1
Multimedia Files	Overwrite	21.6
System Files	Overwrite	21.9
User Documents	Overwrite	11.5
Installation Media	Overwrite	5.1
Log Files	Append	1.6
Mails	Overwrite	1.6
Database Files	Overwrite	1.6

Table 3.1: *The set of file types appearing in the collected userTrace, their conjectured append behaviors, and their contribution percentages in terms of size.*

4KB block in that file, and recorded into database its last modification time and all fingerprints computed this way.

Our tool tracks the file-level changes between consecutive versions of a file by comparing their constituent fingerprint sequences. It assumes that when a file is modified, the entire file is over-written. However, in some cases, modifications to a file could be in the form of appends. Because we had no way of knowing how applications actually modified files, we relied on the file type information to infer whether a file was overwritten or appended at the block level when its file-level change indicated an append-like pattern.

Table 3.1 shows the list of file types appearing in the raw collected trace, their contribution percentage in terms of size and our conjectures of whether they were overwritten or appended at the block level. VM-related files include files that support virtual machines, e.g., vmdk, vmem and vdi files. Multimedia files include all audio and video files. System files include files in the system directory, including Windows and "Program Files" directories. User documents include Microsoft Office files, pictures, and development files. Installation media refer to those files that are meant to install programs, e.g., iso and msi files. Log files include system log files and application log files. Mails include files that are updated by Microsoft outlook, including files with the suffix pst and ost. Database files cover all database files used by either applications or the operating system. Eventually we decided to remove VM image-related files because we believe these files are relatively rare in a typical office environment. After all necessary pre-processing, we produced a trace of daily fingerprint streams, which was initially 10.8GB in size, and eventually grew to 43.7GB at the end of the trace collection period. From the collected trace, the size of the average daily increment change is about 1.5% of all the

files on these 23 machines taken together. However, in this trace when a file is modified, the delta consists of not only the changed data but the entire file data. Hence the deduplication ratio is much higher than a typical block level incremental snapshot.

To measure some precise characteristics of *Sungem* we use another trace which is created based on the idea proposed by Fanglu et al. [69] as follows: Create a vm image called $vmImageBase$ using Vmplayer and use Windows 7 as the guest OS. To $vmImageBase$ add language packs and some huge windows applications to generate another image $vmImageModified$. The measured deduplication ratio in these traces are 20% in each trace. But when they are passed as input one after the other, deduplication ratio is around 34%. Meaning, around 14% of duplicates were found in modified vm image with respect to original vm image. But with this trace, the amount of data available is too short to test various features of a deduplication engine. Hence we use this trace sparingly only when the amount of input data doesn't influence the analysis. Lets call this trace as $VmTrace$.

While analyzing the $userTrace$, we noticed several interesting data sharing patterns that eventually helped make better design decisions in *Sungem*. Interestingly, the finer details of the trace analysis not only helped fine-tune *Sungem* but also looked promising to be applied to any generic deduplication algorithm. Towards that goal, we collected yet another trace using more machines, more users and collected lot more information from the user machines over an even larger period of time, to do a deeper analysis of the trace workload. With such a detailed trace analysis we proposed several guidelines to help design any generic deduplication algorithm and are discussed in greater detail in Chapter 4. The core characteristics of the trace workload that we wish to exploit and demonstrate in this evaluation study do not differ much between the $userTrace$, and the bigger and latest trace that we just described. Hence we just use $userTrace$ and $VmTrace$ in this evaluation study.

The hardware testbed used in the evaluation of the *Sungem* prototype consists of two quad-core 3.4GHz Intel Core i-7 processor, 14GB of RAM, five 7200-RPM WD Caviar blue hard disks of 1TB each, one for the system disk, two for storing the GC metadata P-array on a striped software RAID with a 64-Kbyte stripe unit size, and the remaining two for storing the deduplication fingerprints on a striped software RAID.

3.5.2 Overall Performance

We fed into the *Sungem* prototype with the $userTrace$ spanning 6 days that consists of more than 2 billion fingerprints, measured the deduplication throughputs and ratios, and observed that *Sungem* was able to consistently deliver

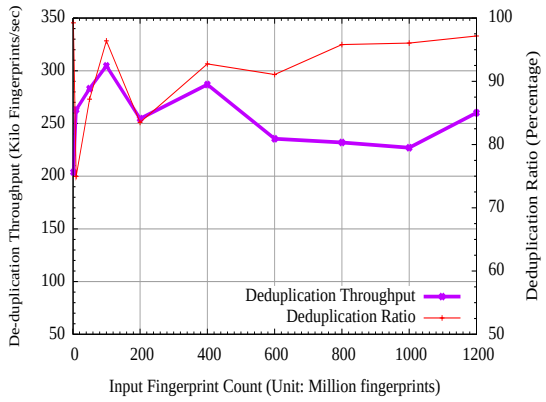

Figure 3.3: *The deduplication throughput and deduplication ratio of the Sungem prototype over an input trace of 2 billion fingerprints*

throughput above 200K fingerprints/second while maintaining a very high deduplication throughput of 90% as shown in Figure 3.3. The throughput of *Sungem* is usually around 250K fingerprints/second but the average comes down because of occasional long pauses caused by Java garbage collector. Java garbage collector maintains separate heap pools for short lived objects and long lived objects, which are called young generation and old generation, respectively. Applying the "Young Objects Die Young" assumption, Java garbage collector attempts to reclaim free memory in the young generation objects more frequently than that in the old generation ones. Efficient java programs tend to keep their objects short-lived. The *Sungem* prototype embraces this rule too, but at times when the container cache cannot capture the working set, more disk I/Os occur, making some objects long-lived and thereby pulling down the overall deduplication throughput. The dips in the deduplication throughput curve in Figure 3.3 arise precisely because *Sungem's* working set at that instant exceeds the container cache. When a fingerprint segment hits an existing fingerprint segment, *Sungem* actually needs to do more work because it needs to bring in a container and performs fingerprint-by-fingerprint comparisons. When a fingerprint segment does not match any existing fingerprint segment, its fingerprints are filtered out by SFI and all subsequent steps in *Sungem* are skipped. Therefore, when the deduplication ratio of an input trace at an instant is higher, its deduplication throughput at that instant should be slower because of additional work. However, the correlation between deduplication ratio and deduplication throughput is not particularly strong in Figure 3.3 for two reasons. First, pauses caused by Java garbage collector have a non-trivial impact on the deduplication throughput and can happen any time. Second, *Sungem* effectively cuts down the disk access cost

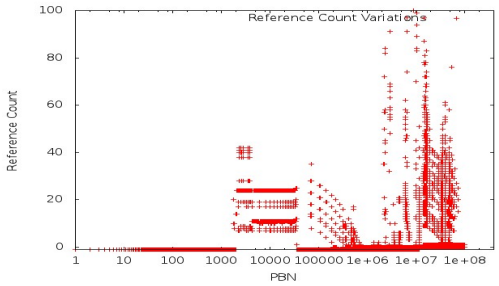

Figure 3.4: *The distribution of reference counts in the input trace. The X-axis is the physical block number (PBN) of each physical block in the backup system and The Y-axis shows how many times a physical block is referenced by different logical blocks.*

of containers when incoming fingerprints hit in the SFI and thus reduces the total overhead in the fingerprint hit case.

We also compared *Sungem's* deduplication ratio with that from the baseline deduplication implementation, which uses a simple hash table to detect duplicates for fingerprints in an input backup stream. In all cases, the absolute difference in deduplication ratio is around 5%. This result shows that *Sungem* did not sacrifice deduplication ratio so as to deliver high deduplication throughput.

If an input backup trace contains only a small number of distinct fingerprints, the throughput of the deduplication engine is naturally high because of high access locality for fingerprints. To demonstrate this is not the case for the input backup trace used in this study, for each input block to the deduplication engine, we either increment its reference count or increment the reference count of its duplicate stored copy. Figure 3.4 shows that the reference count of each physical block in the backup system after *Sungem* traverses the *userTrace* trace is no more than 100 and that the number of unique fingerprints in the input trace is huge. Besides, we also measured the disk activity during the test run and found that the disk was heavily used at all times. The above two evidences prove that the high deduplication throughput of *Sungem* in Figure 3.3 is not because the working set of the input trace is small. As another measure, we tweaked the *userTrace* to produce synthetic traces with deduplication ratios ranging from 0.08% to 95% and *Sungem* delivered a consistent high throughput above 200K fingerprints/second for all the cases.

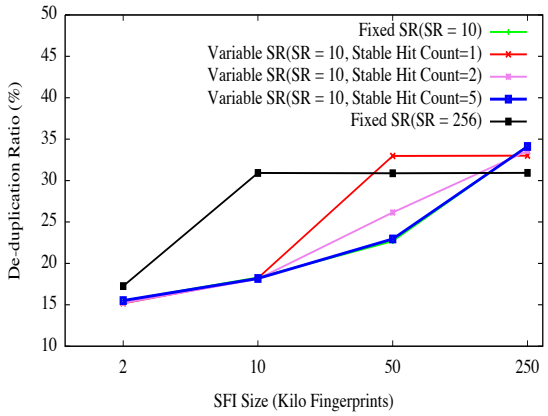

Figure 3.5: *Impact of the size of Sampled Fingerprint Index (SFI) on the deduplication ratio*

3.5.3 Effectiveness of Sampled Fingerprint Index

With *vmTrace*, we varied the sampling rate (SR) and SFI size and measured the deduplication ratio for the following 5 configurations:

1. Fixed SR of 10, i.e., one out of every 10 fingerprints in each stored fingerprint segment is inserted into the SFI.
2. Fixed SR of 256 (each input segment has 256 fingerprints), meaning at all time only 1 fingerprint of each stored fingerprint segment is inserted into the SFI.
3. Variable SR configurations start with an SR of 10 and then switch to an SR of 256 when the number of hits reaches the stable hit count, which is set to 1, 2 or 5. Stable hit count corresponds to the number of hits a stored fingerprint segment needs to experience before it is considered stable and and every sample except one is removed from the SFI.

These five configurations correspond to five different ways of using the memory space allocated to the SFI. Figure 3.5 shows the deduplication ratios of the five configurations when run against the *vmTrace* trace with varying SFI size. We have tried the same experimental set-up using the *userTrace* trace, and the results are similar to those in *vmTrace*. When SFI size is 250K, fingerprints in the SFI are rarely replaced and the more space-consuming configurations produce better deduplication ratio results. For example, fixed SR of 10 is better than fixed SR of 256, and variable SR with a stable hit count of 5 is better than variable SR with a stable hit count of 1. On the other hand, when SFI size is 2K, fingerprints are replaced so frequently that the least space-consuming configuration wins out, i.e., fixed SR of 256.

When SFI size is 10K, fixed SR of 256 is better than all other configurations by a large margin, because the SFI space is too small to hold all the relevant fingerprints in other configurations. However, when SFI size is 50K, number of replacements in SFI were neither too less nor too small and that reflects the desirable scenario in a real world setup. In that case, variable SR with a stable hit count of 1 actually produces higher deduplication ratio than fixed SR of 256, because the former makes the most efficient utilization of the SFI space.

Fixed SR of 256 works surprisingly well across all SFI sizes tested. This suggests that most stored fingerprint segments in *Sungem* that see repetitions could be successfully located when only one of their fingerprints is put into the SFI.

3.5.4 Content Proximity-Based Fingerprint Placement

Sungem strives to place fingerprint segments that share common fingerprints in the same container, and thus uses a *content proximity-based* (CP) approach to determine which container a stored fingerprint segment should be stored. In contrast, other deduplication systems [6, 37] use a temporal proximity-based (TP) approach in that they place stored fingerprint segments that are temporally close in their creation time into the same container. The TP approach to fingerprint segment placement is similar to a write-optimized file system, e.g. log-structured file system, because new stored fingerprint segments are simply appended to the default container until it becomes full. The CP approach to fingerprint segment placement is similar to a read-optimized file system, e.g., UFS, because it tries to place in the same container stored fingerprint segments that are likely to be referenced together when checking an input fingerprint segment against the fingerprint database. Therefore, we expect the TP approach to incur fewer disk write I/Os (for persisting containers) but more read disk I/Os (for determining if an input fingerprint hits the fingerprint database) than the CP approach. The *fill-up threshold* in the CP approach specifies the degree of fullness (in terms of percentage) of the default container before it is considered filled up. The residual capacity of a container in the CP approach is reserved for accommodating future stored fingerprint segments that share common fingerprints with those fingerprint segments already in the container. The TP approach actually corresponds to the CP approach with the fill-up threshold set to 100.

Table 3.2 shows a deduplication ratio and throughput comparison among variations of the CP scheme, each corresponding to a distinct fill-up threshold, when the input load is a 3-day trace consisting of 473 million fingerprints and the container cache size is 5000 containers. Surprisingly, the TP scheme beats

Fill-up Threshold	Dedup Ratio	Dedup Throughput	Container Read Count	Container Write Count	Per-Segment Comparions
70%	93.11%	282.9K	1.238	0.0743	755
80%	93.17%	290.7K	1.248	0.0739	814
90%	93.14%	288.9K	1.259	0.0733	809
95%	93.16%	287.2K	1.267	0.0733	807
100%	93.26%	295.8K	1.264	0.0732	601

Table 3.2: *Deduplication ratio and throughput (fingerprint look-ups per sec) comparison among variations of the content proximity-based fingerprint segment placement approach, each corresponding to a distinct fill-up threshold, when the container cache size is 5000 containers. The TP approach corresponds to the CP approach with the fill-up threshold set to 100%.*

all CP variants with different fill-up thresholds in terms of deduplication ratio and throughput. To understand why, we also measured the average number of distinct container reads and writes when processing a 256-fingerprint input fingerprint segment. As expected, the average number of container reads per input segment increases with the fill-up threshold, but the average number of container writes per input segment decreases with the fill-up threshold. However, the differences are too small to matter. Therefore, the deduplication throughputs across all fill-up thresholds are quite comparable.

Even though the CP variants with less than 100% fill-up thresholds offer the flexibility of clustering related stored segments, the TP scheme could provide the same clustering benefit if the consecutive temporal distance between instances of the same fingerprint sequence is smaller than the container size. However, in the input trace we used, the average temporal distance between consecutive instances of the same fingerprint sequence is actually much larger than the container size. Because the TP scheme tends to fill up a container before switching to a new container, it uses fewer containers and thus incurs a proportionally smaller number of container accesses. In addition, the average number of fingerprint comparisons per input segment required by the TP scheme is noticeably smaller than that required by the CP variants, as shown in the last column of Table 3.2, because the TP scheme is more capable of homing in to the matching stored segments without wasting unnecessary efforts exploring related candidate stored segments. Finally, the target scenarios that the CP scheme is optimized for, i.e., multiple existing stored segments that repeatedly appear together, simply is not very common. The above three reasons combined make the TP scheme the most performant under the input trace used in this study.

3.5.5 Garbage Collection Overhead

Effectiveness of our hybrid Garbage Collection scheme can be demonstrated by comparing it with a vanilla p-array update implementation, which buffers p-array update requests in a queue, and uses a background thread to commit them on a first come first serve basis. In Table 3.3, the throughput of the data deduplication engine without any p-array updates (the last column) sets an upper bound because it corresponds to a zero-cost p-array update scheme. When the BOSC-based p-array update scheme uses a single commit thread, the end-to-end throughput of the deduplication engine is decreased to 19% of the upper bound. By increasing the number of commit threads to 4 and therefore the disk I/O concurrency, it increases the end-to-end throughput to 97% of the upper bound. The number of commit threads represent a tradeoff between disk access locality and disk I/O concurrency. Empirically, the optimal number of commit threads for our experiment set-up seems to be 4. However, regardless of the number of commit threads used, the end-to-end throughput of the data deduplication engine using the vanilla p-array update scheme never exceeds 5% of the upper bound.

Commit Threads	*deduplication + vanilla GC*	*deduplication + BOSC-based GC*	*deduplication + without GC*
1	5879	54047	287204
2	6003	268218	287204
4	9858	277670	287204
10	8121	269272	287204

Table 3.3: *End-to-end throughputs(fingerprints processed/second) of a data deduplication engine with multiple garbage collector configurations.*

To directly assess the effectiveness of the proposed garbage collector, we measured the performance overhead of bookkeeping the reference count and expiration time data structures in the P-array for a series of daily incremental backup runs, each of which uses a delta list of fingerprints as input, and the results are shown in Table 3.4. The "No. of Records" column shows the number of records in the delta list of each day. The "No. of Entries" column shows the number of P-array entries that need to be modified. The "No. of Pages" column shows the number of P-array pages that are to be modified, where each page is 128KB. The "Bookkeeping Throughput" column is calculated by dividing the number of P-array pages to be modified by the bookkeeping time.

Day in Trace	No. of Records	No. of Entries	No. of Pages	Bookkeeping Time	Bookkeeping Throughput
1st	126 M	345 M	185282	71 s	334 MBps
2nd	345 M	917 M	799090	429 s	238 MBps
3rd	344 M	921 M	628042	1017 s	79 MBps
4th	317 M	852 M	597386	1089 s	71 MBps

Table 3.4: *The number of P-array entries modified as a result of the delta list of each day of the input backup trace and the associated bookkeeping time required to put these modifications to disk, when the number of bookkeeping threads is 10*

The fact that the number of P-array entries modified is indeed proportional to the number of records in the delta list shows that the proposed hybrid garbage collection algorithm is indeed scalable. The bookkeeping throughput decreases over time because there are more physical blocks in the data backup system as time goes by and the locality of the updates to the P-array become worse. However, the fact that the bookkeeping throughput remains relatively high demonstrates the effectiveness of the BOSC scheme in turning the random updates to the P-array into largely sequential disk reads and writes.

3.5.6 Effectiveness of Container Cache

Deduplication throughput is heavily influenced by the container cache size because large container cache could capture the working set of containers when processing input fingerprints and reduce the number of container-related disk I/Os. Figure 3.6 shows that for the given workload, the deduplication throughput is poor when the container cache size is below 3000, because the container cache cannot capture the active working set. However, as the container cache grows beyond 3000 containers, the deduplication throughput shoots up and stays flat even when the container cache grows to 5000. This suggests that for the given workload, a cache of 3000 containers is sufficient.

3.5.7 Impact of Controlling Stored Segment Formation

As explained in subsection 3.2.2, the K-factor controls the number of segments in which a fingerprint can participate and acts as a tradeoff between deduplication throughput and ratio. Higher K value allows *Sungem* to identify the best stored segment of maximal length and thereby reduces any additional work required to deduplicate the remaining fingerprints. But at the same time, it also requires extra effort in going over all possible matching stored segments

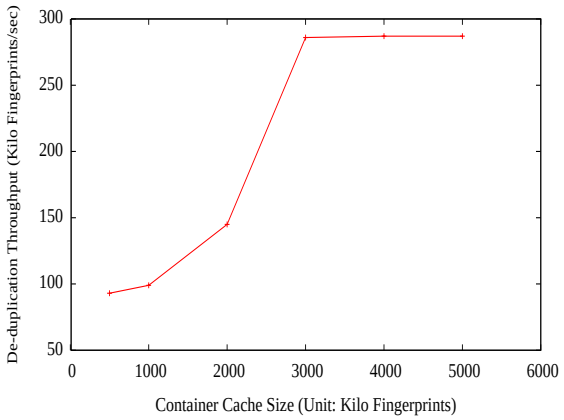

Figure 3.6: *Impact of the container cache size on Sungem's deduplication throughput*

to find the best match. From Table 3.5, K value of 4 gives the best possible throughput and hardly sacrifices on the deduplication ratio.

K-factor	*Throughput*	*Ratio*
1	233	90.90
2	282	92.68
4	287	93.20
5	280	93.21
10	20	93.26

Table 3.5: *Impact of K-factor variations on Sungem's deduplication throughput and ratio. Throughput is measured in Kilo fingerprints/second and Ratio as a percentage of Duplicate fingerprints/Input Fingerprints. Settings include cache size=5000, fillup-threshold=95%*

3.5.8 Parallel Deduplication tradeoffs

Section 3.4 explains two different parallel versions of *Sungem* which performs better than the other depending on the type of input workload. To induce disk boundedness or cpu boundedness into *Sungem*, we will vary the container cache size and then show the difference in deduplicate throughput of the parallel model against a standalone model.

Figure 3.7 shows how the deduplication throughput of Standalone deduplication drops when container cache size is brought down. With *userTrace*

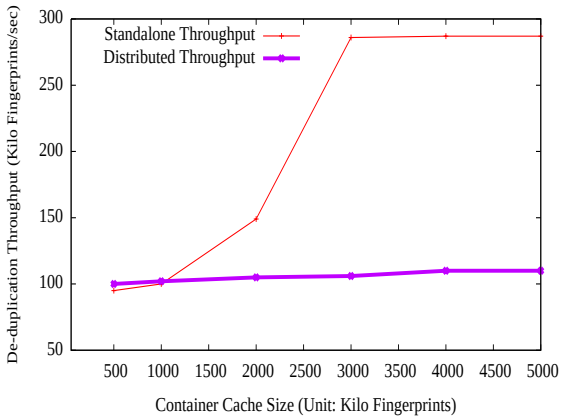

Figure 3.7: *Deduplication throughput comparison between Parallel and Standalone Deduplication*

workload, with cache size less than 500, the system is disk bound because the rate at which containers get evicted from cache is higher than the rate at which they can be stored on disk. When the system is disk bound, standalone deduplication is bottlenecked in loading and storing containers, thereby bringing down the throughput to below 100K fingerprints/second. Hence parallelizing using naive parallel design will not improve the performance as it will only cause more disk I/O's and deteriorate the performance. But the performance of the parallel design with two slave nodes and one master node doesn't deteriorate because when one slave node is bottlenecked by disk I/O, the other slave node continues to utilize the CPU efficiently. Both slave nodes are bottlenecked by disk I/O when cache size is seriously low but that's not the point we are trying to prove. We see that when cache size is more than 3000, the working set of containers are cached appropriately and hence standalone deduplication drives a high deduplication throughput of more than 250K fingerprints/second. The experiment demonstrates the effectiveness of different parallelizing strategies under different conditions. However due to practical limitations, we couldn't test this with a large scale setup consisting of hundreds of slave nodes and this is an important part of our future work.

3.6 Summary

DISCO performs incremental backup operations at periodic intervals and to minimize the data transfer between the primary storage to backup server, DISCO uses block-level deduplication. Typical deduplication techniques bot-

tleneck on disk I/Os involved in various components of the deduplication system, and the most important of them are, a) looking up in the fingerprint index, b) loading and storing containers that contain the possible duplicate fingerprints, c) updating metadata of each disk block after identifying the duplicates. We, therefore built *Sungem* to minimize disk I/Os in the entire deduplication process, and is designed specifically to work efficiently with incremental backup operations. In particular, the following contributions are made:

- An integrated deduplication engine design that combines variable-rate fingerprint sampling and content-based fingerprint-to-container assignment to make the best use of the given memory space and raw disk bandwidth,
- A scalable deduplication engine that delivers consistent high throughput across all ranges of dedupe ratios and improves the deduplication throughput by up to 40% without sacrificing the deduplication ratio, when compared with the state-of-the-art sparse-indexing scheme [6] running with the same amount of RAM, for incremental backup operations,
- The first known garbage collection algorithm whose bookkeeping operations are distributed over individual backup operations and which is scalable in the sense that its bookkeeping overhead for each backup operation is proportional to the change to a VD between consecutive backups rather than the VD itself, and
- A comprehensive evaluation of the deduplication and garbage collection engine with a real-world daily backup trace spanning 10 weeks.
- A study of two parallelization strategies for data deduplication/garbage collection, in terms of the trade-off between communications overhead and amount of redundant disk I/O.

Chapter 4

A Trace-based Study for Deduplication Algorithm Design

Modern data deduplication algorithms are all built on a set of common fundamental techniques based on sampling and prefetching, which were discussed in Chapter 3 in great detail using *Sungem*. For an incremental backup, its safe to assume a duplicity of 50%. It means there is a good probability of an incoming data block for not being a duplicate and hence the majority of fingerprint database look-ups result in a miss, i.e., doing all the work of duplicate check for nothing. Therefore, an important design issue in building modern deduplication engines is how to identify and ignore non-recurring stored fingerprint segments. It would be ideal if a deduplication engine is able to recognize non-recurring fingerprint segments as they appear, and avoid putting them into the in-memory index and fingerprint containers. The next best thing is to quickly recognize them after they are put into the in-memory index and fingerprint containers, and remove them from these data structures. How to identify unproductive stored fingerprint segments and recycle the storage space allocated to them has never received its due attention.

To explore the design trade-offs in deduplication engine design, we took a trace-based approach, in which we collected backup traces and analyzed the detailed behaviors of the deduplication engine when fed with these backup traces. With a good understanding of the real-world trace workload, we fine-tuned *Sungem* to extract better deduplication throughput and deduplication ratio. However, we also noticed several interesting data sharing patterns in the real-world trace that looked promising to optimize any generic deduplication algorithm and not just the *Sungem*. Towards that goal, we collected yet another real-world trace using more machines, more users and collected lot more information from the user machines over an even larger period of time, and did a thorough classification and analysis of the data sharing patterns, which

provides interesting insights to how users share data among them. Finally, to simplify the task of collecting incremental block-level backup traces, we devised a novel trace conversion scheme that is capable of converting file-level changes to block-level changes, and eliminates the need for a kernel-level dirty block tracking agent. This scheme greatly simplifies the backup trace collection task of this work. Throughout this chapter, we refer to a deduplication system in general unless explicitly addressed for *Sungem*.

4.1 Trace Collection and Conversion

Modern block-level data backup systems are incremental in nature and capture the delta between consecutive backups of a disk volume by installing a dirty block tracker in the backup client, which could be a file server, a database server or an end user machine. Our goal is to record an incremental block-level backup trace for the local data volumes of a set of end user machines over a period of time. One way to collect such data backup traces is to install a kernel-level dirty block tracker on every end user machine to be monitored. However, this approach is too intrusive and its performance overhead is too high to be considered a feasible option. Instead, we resort to an approximation approach that is capable of deriving a block-level trace from file-level changes.

4.1.1 Trace Collection

We wrote a user-level trace collection program that periodically scans specific directories at a specific time every day to detect files that are modified since the previous scan. This program was written in Java so that it is easily portable across end user machines that run Linux, Windows and Mac OS. For every directory scanned, the trace collector maintains a timestamp and uses it to detect those files whose last modified timestamp is more recent than their parent directory's modified timestamp. For every modified file, the trace collector records a 64-byte SHA-512 hash value for each block in the file, the file length, the file type, the encrypted file name, the encrypted machine name and the encrypted user ID. Some of the recorded information were encrypted to preserve user privacy. To facilitate easier analysis of the trace data and to isolate data corruption, data and metadata were stored separately for each machine and for each day in a dedicated storage server which held all the trace data safely in a RAID disk array. The trace collector was installed on around 90 desktop machines, which are used predominantly by software engineers in a research organization, and 260 GB worth of raw trace data were collected over a period of 8 months. 78 of these 90 machines run Windows, 11 Linux and 1

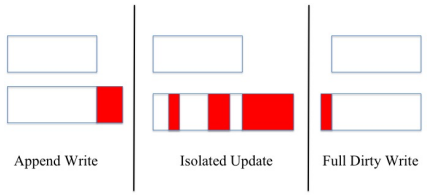

Figure 4.1: *Three types of updates to an existing file to form a new file*

MacOS. Users were given the option to select the location on their machine for trace collector to scan, so that they could exclude some folders containing private data. On rare occasions, the trace collector scanned directories in parallel to user activity and hence as expected the data collected was out of sync with the metadata and hence such data were discarded. The discarded data accounted to less than 1 GB of data which corresponded to 0.79% of the overall data. During the 8-month period, some machines were shut down, and new machines were added. Typically on Windows machines the C: directory was excluded and on Linux machines only the /home directory was included. Majority of the machines were configured either for Linux or Windows only and only a few had dual boot options but were never exercised during the trace collection period.

4.1.2 Trace Conversion

The trace collector records file-level changes, but our goal is to capture block-level changes and turn these changes into an incremental backup trace via a trace converter. Instead of treating every block in a modified file as a dirty block appearing in an incremental backup trace, the trace converter classifies updates to a file into the following three types, as shown in Figure 4.1:

- *Appended write*: If the N+1-th version of a file is equal to a concatenation of the N-th version with an additional sequence of K blocks in the end, then the trace converter considers only the additional K blocks as dirty blocks, because it assumes these blocks are appended to the N-th version to form the $N+1$-th version.

- *Isolated update*: If the $N+1$-th version of a file is not an extension of the N-th version through appends, the trace converter considers every block in the $N+1$-th version as a dirty block, except those blocks in the $N+1$-th version that are identical to their counterparts in the N-th version in terms of both their content and their offset within the file, because it assumes the updates to the N-th version are isolated block-level updates, rather than whole file overwrites.

- *Full dirty write*: If the $N+1$-th version of a file results from pre-pending a piece of new data to its N-th version, because the two versions do not have the same content at any given offset, the trace collector counts every block in the $N+1$-th version as a dirty block.

The raw input trace (preprocessed file level trace) consists of 24.1 million files and 3.97 billion blocks, of which 18.3 million files are unique and the remaining 5.8 million files are versions (modifications) of some existing files, according to their file names. Appended write files account for just 69 files with 340K blocks in them, of which 159K blocks represent common blocks between consecutive file versions, and thus are filtered out from the resulting incremental block-level backup trace. Interestingly 4.1 million files match perfectly with their previous versions and consist of 65.5 million blocks. These files were considered modified by our trace collector simply because of changes to their file metadata such as read/write permission and last modified time. In addition to perfectly matched files, there are 363K files that share at least one common block with their previous versions and consist of 2.7 billion blocks, among which 2.48 billion blocks are found to be identical to their counterparts in their previous versions and thus are also excluded from the resulting block-level trace.

Once dirty blocks are identified, the trace converter assigns a logical block number (LBN) to every block of every modified file. To facilitate the process of locating the file that contains a given block, the trace converter assigns a continuous range of LBNs to the blocks that belong to versions of the same file. This way, given a LBN, one could locate the containing file with a table look-up. To achieve this, the trace converter first went through the raw input trace to derive the maximum file size for every file appearing in the trace, and then assigned every file a range of LBNs equal to its maximum file size. This two-phase approach to LBN assignment greatly decreases the amount of disk access overhead that would have been required to implement one LBN range per file given the raw input trace is 260 GB in size. As a result, using a commodity desktop machine, we could complete the two-phase trace conversion process within a few minutes. The eventual output of the trace converter is a block-level backup trace that consists of entries each of which is of the form <LBN, fingerprint> and corresponds to a modified block.

4.1.3 Trace Processing

Given a block-level backup trace, we need to first establish the ground truth for how the fingerprints in the trace match one another. Towards this goal, we built a vanilla deduplication engine called *Baseline*, which processes the

input trace in two phases. In the first phase, *Baseline* partitions entries in the input trace into buckets, each of which is stored as a file, then applies a hash function to the fingerprint of every trace entry, and finally assigns a trace entry to a bucket if the hash value of its fingerprint falls into the bucket's associated range. This partitioning ensures all duplicates are assigned to the same bucket. Each bucket's size is set to be slightly smaller than the available main memory. In the second phase, *Baseline* processes each bucket one by one, and when processing a bucket, it reads in trace entries in the bucket and builds a fingerprint database for these entries on the fly to perform duplicate check. Because the way the bucket size is chosen, the resulting fingerprint database is guaranteed to fit within main memory and the associated duplicate checks proceed without incurring any additional disk I/O beyond reading the input trace entries. As a result, *Baseline* is able to complete the duplicate checks for the 260-GB trace, orders of magnitude faster than any naive method. A given input trace entry may match multiple previous trace entries and *Baseline* prioritizes the matched duplicates in the following order:

1. Choose the duplicate with the same file ID as the input trace entry's file ID.
2. Choose the duplicate that belongs to a file that contains a match to any of the last 256 entries in the original input trace.
3. Choose the duplicate whose creation timestamp is closest to the creation timestamp of the input trace entry.

Priority 1 corresponds to the scenario where a match is found within a file or with one of its versions. Priority 2 corresponds to the scenario where any of the last 256 matches in the original input trace (first phase) fetched in a file and that file contained a match for the given input trace entry. There is a good chance that the file is cached and available for subsequent matching and hence *Baseline* prioritizes such a match over a random match. Number 256 is just a heuristic and has no particular resemblance to anything in the algorithm or the trace. Priority 3 corresponds to the scenario where a match is seen in recent past and hence is cached for subsequent matching. When an input trace entry in the second phase matches an existing entry in the fingerprint database, the match result is recorded in an entry of a match table. Every match table entry contains the following information about both the matched and input trace entries, the file ID, file length, machine ID and OS type.

File Type	Extensions
VM_related	vmdk, vdi, vmem, img, vmss, iso, vhd
Image	JPG, gif, bmp, tiff, png, IMD, psd, psp, gis
Compress	zip, tar, gz, rar, 7z, cab, tgz, bz2, jar, cfs
Database	sql, dat, MYD, sdf, db
Audio_Video	avi, mov, mpg, mpeg, mp3, wmv, wav, mp4, idx, flv, mts, flac, wmdb, mkv
Document	txt, rtf, doc, xls, pdf, ppt, html, xml
Program	c, cpp, java, php, js, svn
System	rpm, dmg, exe, dll, msi, ipsw, so, ipa, hdmp
Outlook	ost, pst, oab
Log	log
Others	remaining file types

Table 4.1: *Each file type corresponds to a class of files that share some common semantic property and encompass a set of file name extensions.*

4.2 General Duplicity Pattern

We first categorize files in the input trace into the file types listed in Table 4.1 according to their file name extensions. Files that do not belong to any listed file type are labeled as *Others*.

Each row in Table 4.2 shows that, for a particular file type, the percentage of blocks in the input trace that belong to that file type, the percentage of blocks in that file type that turn out to be a duplicate, and the ratio between the second and third columns. The *Input Block %* column represents the deduplication cost because every block costs something to store and to check for duplicity, and the *Overall Duplicity %* column represents the deduplication gain. The fourth column thus indicates how worthwhile it is to include a particular file type in deduplication check. The higher a file type's cost ratio, the less desirable it is to include the file type into deduplication. According to this metric, program file type is the most promising candidate for deduplication, whereas audio_video file type is the least desirable.

VM related files make up a large majority of the input blocks (38.97%) and an even larger majority among the duplicated blocks (51.41%). This is expected because these type of files are much larger in size and the majority of their blocks are shared unmodified. Although program files contribute to only

File Type	Input Block %	Overall Duplicity %	Cost Ratio
VM_related	38.97	51.41	0.76
Image	2.76	2.42	1.14
Compress	11.79	10.23	1.15
Database	6.9	9.56	0.72
Audio_Video	4.7	1.57	2.99
Document	1.31	0.91	1.44
Program	1.49	2.46	0.60
System	4.09	3.43	1.19
Outlook	2.84	1.96	1.45
Log	1.28	0.84	1.53
Others	23.88	15.21	1.57

Table 4.2: *Percentages of blocks and duplicated blocks that belong to a variety of file types*

1.49% of all input blocks and 2.46% of all duplicate blocks, a large majority (84.69%) of the blocks in program files are duplicates, and at the file level, the duplicity percentage for program files is even higher, at 86.65% (explained later in Table 4.8).

Though files tagged as Others were regarded as inconsequential because of their low representation in the input workload, collectively they contribute to a majority of the duplicates. Even upon filtering out VM related files, their representation stayed below 1% of the overall input workload and hence they continue to be tagged as "Others". Also since their characteristics are inconsequential from deduplication point of view, we will continue to ignore their results.

In Table 4.3, the second column shows the percentage of blocks that match blocks of the same type and the third column shows shows the percentage of blocks that match blocks of any type. The fact that the difference between these two columns is generally small suggests that most of the duplicates are found in files of the same type, except for program files. Only 3.97% of the duplicates have their source and target from different file types. A deduplication engine could take advantage of this fact to avoid unnecessary fingerprint comparisons by organizing fingerprints into separate indexes, one for each major file type.

In Table 4.4, each row corresponds to blocks belonging to files of different file size ranges and shows in the second column, the percentage of blocks in the input trace, in the third column, the duplicity as a percentage of all the

File Type	Duplicity % with same Target File Type	Duplicity % with any Target File Type
VM_related	64.07	67.53
Image	40.38	44.84
Compress	41.37	44.41
Database	67.37	70.94
Audio_Video	15.44	17.10
Documents	28.77	35.61
Program	47.23	84.69
System	34.73	42.90
Outlook	34.90	35.42
Log	32.70	33.42
Others	29.24	32.61
Overall	47.21	51.18

Table 4.3: *Percentage of duplicates whose source and target are of the same or different file types*

duplicate blocks, and in the fourth column, the duplicity as a percentage of duplicate blocks that belong to files local to the same row.

Though large (>100KB size) files account for 45.29% of the input blocks, only 8.15% of the blocks in large files are duplicates, but these duplicated blocks still represent a fair share (20.39%) of all the duplicated blocks found. On the other hand, small files (1-50 bytes) account for just 6.92% of the input blocks, and yet they contribute to 19.56% of the duplicated blocks and 51.18% of the blocks in small files are duplicates.

Finally, the following duplicity patterns also seem to be interesting and noteworthy:

- 15.08% of the duplicate blocks are found within the same file and 4.15% of the duplicate blocks are found in different versions of the same files.
- 12.19% of the duplicate blocks belong to identical files residing on the same user machine. These duplicate blocks account for 48.89% of their associated files.
- 8.59% of the duplicate blocks belong to identical files residing on different user machines. These duplicate blocks account for 36.79% of their associated files.

File Size Range (B)	Input Block %	Overall Duplicity %	Local % Duplicity
1-50	6.92	19.56	51.18
50-100	1.28	2.6	37.19
100-500	5.57	8	26.22
500-1000	3.15	3.85	22.11
1000-5000	10.34	11.16	19.53
5000-10000	3.43	4.34	22.87
10000-20000	3.28	5.15	28.46
20000-50000	7.45	7.6	18.46
50000-100000	13.29	17.25	23.5
>100000	45.29	20.39	8.15

Table 4.4: *Percentage of duplicates found in files of different size*

4.3 Trace-based Deduplication Design Tradeoff Analysis

In this section, we explore the performance impacts of different design choices for deduplication algorithms based on our analysis of the duplicity patterns in the input trace. Specifically, we analyze the following three deduplication algorithm issues: the sampling mechanism for picking and placing representatives of stored segments in the sampled fingerprint index (SFI), the assignment scheme for choosing containers to hold newly formed stored segments, and the reclamation strategy for recycling space held by stored fingerprint segments. In order to explain some concepts, we refer to *Sungem* but it should be noted that the reasoning should be similar to any generic deduplication algorithm. Unlike the multi-threaded implementation of *Sungem* in Chapter 3, *Sungem* is configured here to be single-threaded in all the evaluations used in this trace analysis because of the following reasons: a) the focus is on understanding the design tradeoffs rather than the absolute performance numbers, b) It is possible to measure accurate statistics in a single threaded application c) It is possible to isolate and reason with high accuracy on every important observation in the performance results.

4.3.1 Sampling of Stored Segments

Sungem provides two levels of indexing: SFI relates an input fingerprint to containers that hold fingerprint segments containing the input fingerprint, and within each such container a segment index relates the input fingerprint to those fingerprint segments containing it. Therefore, fingerprints in SFI are like anchors, which point to possible stored fingerprint segments that may match some fingerprint sequence in the input segment. Therefore, it is desirable to put into SFI anchors for as many different stored segments as possible. This argues for coarse sampling for each stored fingerprint segment. On the other hand, if a stored segment is under-sampled, e.g. one in every five fingerprints is put into SFI, a subsequent instance of a fingerprint sequence of length of 4 or less of the stored segment may go undetected, none of its fingerprints is chosen as an anchor.

The current design of *Sungem* is to start by inserting every fingerprint in a new stored segment into SFI, and after a stored segment sees some number of repetitions, the number of representatives for that stored segment in SFI is reduced to one. This sampling strategy is conservative at the beginning and becomes aggressive for fingerprint sequences that prove to be recurring.The only room for further improvement is to choose the starting rate slightly more aggressively. For example, when a new stored segment is a superset or subset of existing stored segments, it is not necessary to insert each of its fingerprints into SFI. Perhaps for the part that overlaps with existing stored segments, its fingerprint samples currently in SFI should be sufficient.

Table 4.5 shows the size distribution of recurring stored segments, i.e., those fingerprint segments that see at least one repeating instance, given the input segment length is 256. The average recurring stored segment size is 63.03. Because the percentage of recurring stored segments that are smaller than or equal to 8 is relatively small, it seems that the starting sampling rate could be set to sample one every eight consecutive fingerprints without noticeably degrading the deduplication ratio.

4.3.2 Placement of Stored Fingerprint Segments

When a stored fingerprint segment is formed, it needs to be placed in a container. There are at least two approaches to determine the target container to which a newly formed stored segment should be assigned as shown in figure 4.2.

The *content proximity-based* (CP) approach places a newly formed store segment in the same container as the first fingerprint of the preceding hit fingerprint sequence if it is a miss fingerprint sequence, or as the first fingerprint of the residing hit fingerprint sequence if it is a match fingerprint sequence.

Size Range	%age Distribution
1 - 1	3.647
2 - 2	0.008
3 - 4	0.010
5 - 8	0.011
9 - 16	24.284
17 - 32	24.156
33 - 64	18.332
65 - 128	14.119
129 - 256	15.431

Table 4.5: *Size distribution of recurring stored fingerprint segments*

Fill-up Threshold	**Dedup Ratio**	**Dedup Throughput**	**Container Read Count**	**Container Write Count**	**Per-Segment Comparions**
30%	52%	65K	0.77	0.55	200
40%	52%	73K	0.77	0.56	200
50%	52%	83K	0.77	0.56	199
60%	52%	88K	0.77	0.56	198
70%	52%	87K	0.78	0.57	195
80%	52%	86K	0.78	0.57	193
90%	52%	90K	0.78	0.57	193
95%	52%	94K	0.78	0.57	190
100%	52%	102K	0.80	0.57	171

Table 4.6: *Deduplication Ratio and Throughput (fingerprint look-ups per sec) comparison among variations of the content proximity-based fingerprint segment placement approach, each corresponding to a distinct fill-up threshold, when the container cache size is 5000 containers. The TP approach corresponds to the CP approach with the fill-up threshold set to 100%.*

In contrast, other deduplication systems [6, 37] use a temporal proximity-based (TP) approach in that they always place stored fingerprint segments that are created around the same time into the same container. The TP approach to fingerprint segment placement is similar to a write-optimized file system, e.g. log-structured file system, because new stored fingerprint segments are simply appended to the default container until the container becomes full. The CP approach to fingerprint segment placement is similar to a read-optimized file system, e.g., Unix fast file system (FFS), because it tries to place in the same container stored fingerprint segments that are likely to be referenced together when checking an input fingerprint segment against the fingerprint database. Therefore, we expect the TP approach to incur fewer disk write

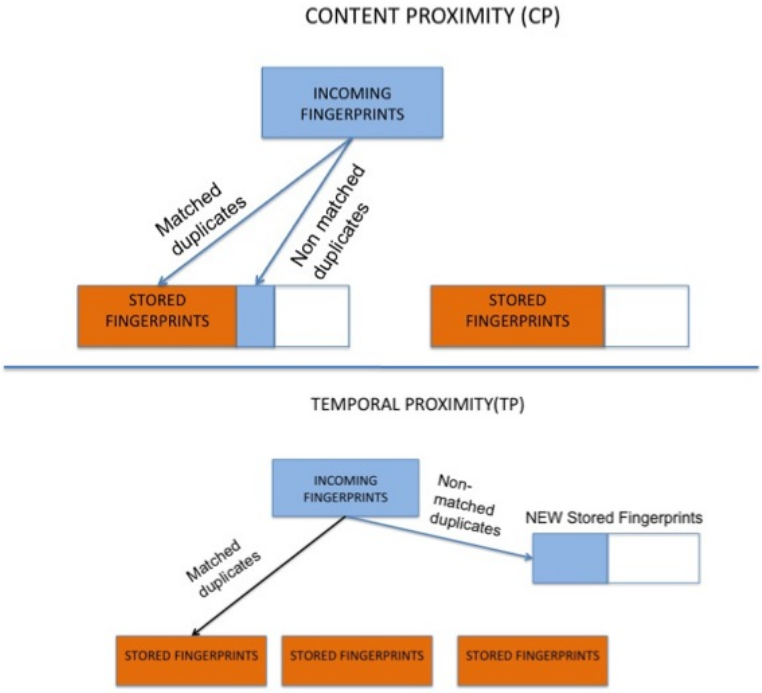

Figure 4.2: *Comparison between TP and CP fingerprint placement strategies*

I/Os (for persisting containers) but more read disk I/Os (for determining if an input fingerprint hits the fingerprint database) than the CP approach. The *fill-up threshold* in the CP approach specifies the degree of fullness (in terms of percentage) of the default container before it is considered filled up. The residual capacity of a container in the CP approach is reserved for accommodating future stored fingerprint segments that share common fingerprints with those fingerprint segments already in the container. The TP approach actually corresponds to the CP approach with the fill-up threshold set to 100.

Although the CP approach puts related stored segments in the same container to reduce the number of containers required to process an input segment, the fact that its containers are mostly less than full means in general it needs more containers to hold the same amount of data than the TP approach. The trade-off between leaving space in each container to cluster related stored segments into the same container and wasting space in each container and thus requiring more containers is very complex. However, by measuring the average number of containers read and written and the number of fingerprint comparisons when processing each input segment, it is still possible to analyze the relative merits of TP and CP under a real-world incremental backup trace. Table 4.6 shows the deduplication ratio and deduplication throughput

comparison among variations of the CP scheme, each corresponding to a distinct fill-up threshold, when the input load is a 6-day trace consisting of 196 million fingerprints and the container cache size is 5000 containers. Surprisingly, the TP scheme beats all CP variants with different fill-up thresholds in terms of deduplication throughput. To understand why, we also measured the average number of distinct container reads and writes when processing a 256-fingerprint input fingerprint segment. As expected, the average number of container reads per input segment increases with the fill-up threshold, but surprisingly the average number of container writes per input segment also increases with the fill-up threshold. A detailed observation showed that on an average only 1.85 number of stored segments are matched for every input segment. As a result, the TP variant isn't affected by the marginal increase in the number of container reads and more importantly, both the TP and CP variants do not differ much in the number of container writes. In the TP approach, the container writes are always applied either to non-full containers or to new containers, both of which are always present in memory. Whereas, in the CP approach, the container writes could happen on any non-full container that has matching fingerprints. Among the different variants, though the differences in container writes per input segment are too small to matter, TP approach performs better than CP variants because the container writes result in random disk IO for CP approach compared to sequential writes in TP approach.

Even though the CP variants with less than 100% fill-up thresholds offer the flexibility of clustering related stored segments, the TP scheme could provide the same clustering benefit if the consecutive temporal distance between instances of the same fingerprint sequence is smaller than the container size. However, in the input trace we used, the average temporal distance between consecutive instances of the same fingerprint sequence is actually much larger than the container size. Because the TP scheme tends to fill up a container before switching to a new container, it uses fewer containers and thus incurs a proportionally smaller number of container accesses. In addition, the average number of fingerprint comparisons per input segment required by the TP scheme is noticeably smaller than that required by the CP variants, as shown in the last column of Table 4.6, because the TP scheme is more capable of homing in to the matching stored segments without wasting unnecessary efforts exploring related candidate stored segments. Finally, the target scenarios that the CP scheme is optimized for, i.e., multiple existing stored segments that repeatedly appear together, simply is not very common. The above mentioned reasons combined make the TP scheme the most performant under the input trace used in this study.

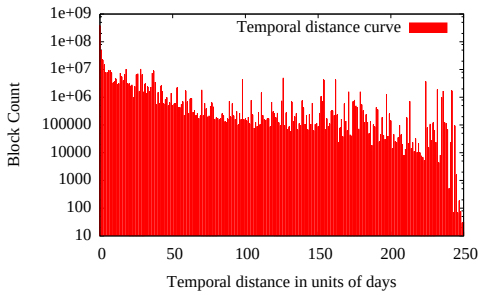

Figure 4.3: *Histogram showing the temporal distance between the matched blocks*

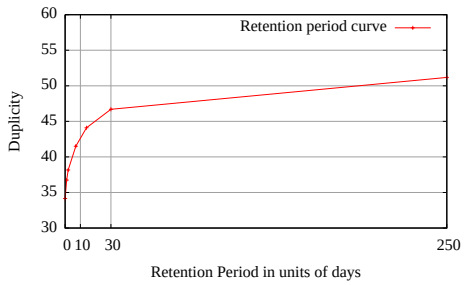

Figure 4.4: *Figure showing the variation in duplicity as retention period of the fingerprints in the lookup database is varied*

4.3.3 Garbage Collection of Fingerprints

The value of fingerprints has a limited shelf time because files that are shared among users over a period of time eventually fade away. Accordingly, as time goes over, the utility of fingerprints for deduplication purpose diminishes and the system should reclaim the resources allocated to them. To get a better understanding of the useful time spans of fingerprints, we measured the temporal distance between the source fingerprint and the target fingerprint of every duplicate found by *Baseline* and Figure 4.3 shows the resulting histogram, where the X axis is temporal distance in terms of day, and the Y axis is the count of duplicates with a specific temporal distance. There are huge spikes when the temporal distance is small, indicating that the temporal distance between the source and target of a duplicate mostly lies between 0 to 30 days, with a significant percentage of duplicates having a temporal distance of 0, i.e., the source and target fingerprints appearing on the same day.

A simple fingerprint garbage collection policy is to throw away fingerprints that are older than a certain age. This could potentially improve the deduplication throughput because more relevant data can be kept in memory, but

Dedup-lication Granularity	All Files	VM related Files	Outlook Files	Log Files
Full File	18.1	8.1	5.5	20.9
1K blocks	37.1	41.2	6.3	21.1
128 Blocks	40.2	45.9	6.6	31.9
64 Blocks	41.3	47.5	13.7	32.8
32 Blocks	42.3	49.6	15.3	33.4
16 Blocks	43.7	52.6	16.2	33.6
8 Blocks	45.1	55.7	16.7	33.8
4 Blocks	46.9	59.4	17.0	34.0
1 Block	51.2	67.5	35.4	34.0

Table 4.7: *Variations in block duplicity across different deduplication granularities for all file types, VM_related, Outlook and Log files*

hurts the deduplication ratio because duplicates with older sources won't be found.

Figure 4.4 shows how the duplicity % varies when fingerprints older than a certain age are excluded from deduplicate checks, and demonstrates convincingly that a drop in duplicity percentage is relatively insignificant when fingerprints beyond a certain temporal distance are deleted from containers to recycle the space allocated to them.

Another interesting related observation from the duplicates found by *Baseline* is that 60% of the files exhibit 1-1 sharing pattern, meaning the blocks in the given two files do not serve as duplicates to any other files. If such file pairs could somehow be identified at run time, their resources in SFI and containers can also be recycled early on.

4.4 Impact of Deduplication Granularity

It is well known that file-level deduplication incurs less checking overhead but produces lower deduplication ratio than block-level deduplication. In this section, we compare file-level with block-level deduplication for different file types using our backup trace.

Table 4.8 shows for each file type its *file-level file duplicity*, which is the percentage of files that are duplicates, *file-level block duplicity*, which is the percentage of blocks that are duplicates and belong to a duplicated file, *block-level block duplicity*, which is the percentage of blocks that are duplicates,

File Type	File level File Duplicity	File Level Block Duplicity	Block Level Block Duplicity	% difference Block Duplicity	Average File Length
VM_related	24.09	8.13	67.53	87.95	270.9K
Image	65.09	34.59	44.84	22.86	15.41
Compressed	67.97	17.61	44.41	60.34	1732.94
Database	12.93	41.91	70.94	40.92	1021.65
Audio_Video	6.70	11.68	17.10	31.67	1388.28
Documents	48.88	24.11	35.61	32.29	19.98
Program	86.65	75.00	84.69	11.44	7.08
System	62.33	34.95	42.90	18.53	653.58
Outlook	81.90	5.52	35.42	84.41	1179.58
Log	10.93	20.90	33.42	37.46	249.66
Others	51.32	19.17	32.61	41.22	22.29
Overall	58.15	18.10	51.18	64.64	61.31

Table 4.8: *The file-level duplicity, block-level duplicity and average file length in units of blocks for each file type*

the percentage difference between *block-level block duplicity* and *file-level block duplicity*, and the average file length.

Intuitively, if files of a file type are rarely modified and/or relatively small, the percentage difference between its *block-level block duplicity* and *file-level block duplicity* should be small. Program files are one such example, because they generally are used as components of a big software project and hence remain unchanged. Because their average file length is just 7.08 blocks, a small number of file modifications do not hurt their overall block duplicity that much. In the end, their percentage drop in block duplicity is the lowest (11.44%). We originally expected that in a research lab where the backup trace was collected, modifications to Audio_Video files should be small and the percentage difference between file-level block duplicity and block-level block duplicity should be negligible. Surprisingly for Audio_Video files, the percentage drop in block duplicity is noticeable, at 31.67%, because the average file size is large at 1388.28 blocks and any change to an Audio_Video file costs dearly in file-level block duplicity. VM_related files are large and expected to experience frequent modifications because of VM image rebuild. Consequently, its percentage drop from file-level block duplicity to block-level block duplicity is the highest, at 87.95%. Document files also experience relatively frequent modifications, but their average file size is just 19.98 blocks. So the drop in block duplicity is smaller than VM_related files, at 32.29%.

We varied the deduplication granularity from one disk block, two disk blocks, all the way to 1000 disk blocks, and measured their corresponding block duplicity, i.e., the percentage of disk blocks that are duplicates and belong to a duplicate chunk.

Table 4.7 shows that the average block duplicity drop for all file types decreases from 18.1% when a file-sized deduplication chunk is used to 51.2% when a block-sized deduplication chunk is used. In addition, it also shows the block duplicity variation across different deduplication granularities for VM_related, Outlook, and Log files. As expected, some file types prefer smaller deduplication chunk size, for example, VM_related and Outlook, but some file type prefers larger deduplication chunk size, for example, Log files.

4.5 To BF or Not to BF

Bloom filter (BF) [172] is a well-known approximate data structure that is specifically designed to facilitate the processing of set membership queries. If a BF answers "no" to a set-membership query, it is guaranteed that the query element is definitely not in the set. However, if a BF answers "yes" to a set-membership query, the query element may or may not be in the set. Existing deduplication systems apply a BF to filter out early on fingerprints that are not in the fingerprint database. *Sungem* supports a Refreshable Bloom Filter (RBF) that allows portions of a Bloom filter to be deleted in an FIFO order, and thus is able to accommodate an indefinitely long stream of fingerprints.

Figures 4.3 and 4.4 suggest that stored fingerprints become less and less useful over time and therefore should be removed from the fingerprint database and the BF as they grow older. Accordingly, an RBF is designed to support this kind of use case, and consists of a chain of M standard BFs each of size $\frac{S}{M}$. Given a query element X, an RBF tests X against every constituent BF, and declares a miss only when all of them report a miss. Initially all BFs in an *RBF* are empty. As new fingerprints fill up a BF, e.g., when its false positive rate exceeds a certain threshold, a new BF is allocated to hold more fingerprints, until all BFs in the *RBF* are used up. At that point, the oldest BF is cleared so that it can be used to hold future new fingerprints. Essentially this implements a coarse-grained FIFO replacement policy. To avoid throwing away relevant fingerprints in the oldest BF, whenever all the BFs in an *RBF* are in use and a new fingerprint hits in the oldest BF, the RBF inserts it in the youngest BF. This last optimization allows old but useful fingerprints to continue to stay in an RBF.

Table 4.9 shows that turning on RBF on the current *Sungem* prototype, which already uses SFI for initial filtering, has no effect on *Sungem*'s deduplication ratio but actually slows down the deduplication engine. There are several reasons for this result. First of all, in general RBF cannot conclude a positive fingerprint match without consulting auxiliary on-disk data structures. When an input fingerprint hits in the SFI, this fingerprint is definitely

RBF status	Deduplication Ratio	Deduplication Throughput	Fingerprint Comparisons
RBF Off	50%	102K	171
RBF On	50%	98K	168

Table 4.9: *Sungem's de-duplication throughput and ratio when* RBF *is turned on and off.*

in the fingerprint database. So SFI guarantees zero false positive rate. In contrast, RBF has non-zero false positive rate. To avoid data loss, zero false positive rate is more important to a deduplication system than zero false negative rate. To avoid false positive, every RBF hit must be followed up with a lookup with an on-disk data structure to determine if the input fingerprint indeed matches some entry in the fingerprint database. Therefore, it is more expensive to ascertain a fingerprint database hit with RBF than with SFI, which does not require disk access.

Second, in the current *Sungem* prototype, RBF look-up is more expensive than SFI look-up, because the former involves multiple hash table look-ups (about 1.5 μs), whereas the latter involves only one hash table look-up (about 0.5 μs). That is, it is also more expensive for RBF to conclude a fingerprint database miss than SFI. Against the input backup trace, RBF and SFI provide comparable negative filtering, i.e., whenever an input fingerprint misses in the RBF, it also tends to miss in the SFI as well. Finally, against the input backup trace, only 44% of the responses from the RBF are "no match." Because the "no match" response percentage is not particularly high, the number of fingerprint comparisons that RBF saves ($171 - 168 = 3$) is not significant enough to justify its higher look-up overhead.

4.6 Summary

There have been several analysis studies of real-world backup traces [60, 66], as well as evaluation studies of specific disk data deduplication systems [64, 65, 67]. This work takes an algorithm design-driven approach to analyzing real backup traces, and focuses on measuring characteristics of backup traces that could reveal the pros and cons of specific deduplication algorithm decisions. Through this approach, we have uncovered several important design dimensions of deduplication algorithms that have not received much attention in the literature. Specifically, this work makes the following research contributions:

- Develop a novel user-level dirty block tracer that is able to collect file-level changes and convert these file-level changes to block-level changes without installing any kernel agent,
- Give the average percentage of duplicate blocks in a file of a certain file type and a certain file size range, and the probability of a pair of duplicate blocks that belong to files of the same name, files with different names but on the same machine, and files with different names and on different machines,
- Show the design tradeoffs for the sampling strategies for fingerprint segments to reduce fingerprint index size, the clustering schemes of related fingerprint segments into containers to improve prefetching effectiveness, and when stored fingerprint segments are retired,
- Quantify the impact of deduplication granularity, and
- Demonstrate that a variant of Bloom filter called RBF is actually harmful when implemented on top of a sampling fingerprint index.

Chapter 5

An Update-Aware Storage System for Low-Locality Update-Intensive Workloads

A low locality disk access workload predominantly consists of random disk I/O requests with a large working set. A well-known technique to boost the performance of workloads dominated by low-locality disk writes is to structure the storage system as a log and convert each disk write into an append to the log. However, such techniques carry hidden performance overheads in the form of metadata look-up and garbage collection. For workloads dominated by low-locality disk reads, no generally effective performance-boosting solution is available, unless the storage system uses expensive hardware devices like flash-based disks. Though, storage systems using flash-based disks typically handle the random read I/O requests quite well, they are known to perform poorly over a workload dominated with random write I/O requests [173].

A disk update request retrieves a disk block, modifies it and writes the resulting block back to the disk. A workload consisting predominantly of such disk update requests, with large working set and low locality, poses severe challenges to traditional storage techniques described above. Such workloads are commonly seen in many real-world applications like user-generated content management and online transaction processing (OLTP) applications. In DISCO, there are quite a number of components that generate such random disk update workloads. 1) The garbage collector (GC) component in DSS (described earlier in Chapter 3) maintains a metadata-index for all the physical blocks in the storage system, and its incoming workload arrives at a very high rate and consists of a large number of disk update requests with very low locality, where each request intends to update portions of the metadata corresponding to some physical block. 2) DMS maintains various disk-based

mapping tables to map large number of blocks in different address-spaces, and its workload consists of a large number of requests that either read an existing entry, write a new entry, or modify an existing entry in the mapping-table. The read/write/modify requests in the workload have very low locality of reference and hence the workload has a large working set.

Traditional storage systems do not perform well under these workloads and thus require higher storage stack layers to carefully schedule incoming disk requests to mitigate the performance penalty associated with such workloads. Even storage systems using flash-based disks perform poorly in the face of these workloads, sometimes faring even worse when compared with magnetic disks. A large body of previous research efforts on disk buffering [174, 175], caching [176–178] and scheduling [179, 180] have attempted to solve this problem, but they are generally ineffective because low access locality leads to excessive random disk I/Os.

Apart from the low locality aspect of the workload, a disk update request is not only challenging but is also interesting because it is fundamentally a disk read followed by a disk write. With the primary motive of delivering high throughput for a workload, a typical storage system services the disk write requests asynchronously using standard buffering techniques, but in order to minimize the response time for disk read requests, it services the disk read requests as synchronously as possible. On the surface it appears that there is not much one could do to boost the performance of workloads dominated by low-locality disk updates, but it is interesting if the storage system can distinguish between standalone disk reads and disk reads associated with disk updates. Concretely, for a disk update request, it's OK to service the update's leading disk read access asynchronously, like its following disk write access, because a disk update is considered completed only after its following disk write is completed. Unfortunately, typical storage systems do not provide such an explicit *update-aware* disk access interface similar to the traditional read/write interfaces.

Traditional storage systems support a disk access interface for higher-layer systems software, such as a file system or a DBMS, to read or write data stored on disks. The granularity of disk reads and writes ranges from disk blocks [181, 182] to more sophisticated constructs such as objects [183, 184]. Regardless of access granularity, these simple read/write interfaces are not adequate for low-locality update-intensive workloads for two reasons. First, there is no update-aware interface to optimally handle the disk-update requests for the reasons mentioned above. Second and more importantly, to optimally handle the disk IO requests that modify the contents of a disk block, the storage system aggregates the requests synchronously into memory and marks that

disk block as dirty. In the background, the storage system sequentially scans the disk, fetches the dirty blocks into memory, applies all the modifications to that disk block and writes the disk block back to the disk. This is very similar to the strategy employed in InnoDB [185] which is a standard storage engine used in most of the MySQL applications. However, this approach fails in certain conditions. When an application issuing disk IO requests requires an ordered collection of data in the disk blocks, it is not straightforward for the storage system to aggregate and apply the modification requests in background because at the time of aggregation, it is not possible to determine the position of modified data on the disk. It is also not possible to determine the position of the modified data within the disk IO block at the time of background commit, because the storage system is generic and oblivious to the applications data placement strategy, and therefore relies on application processes to perform these updates.

To remove the above two problems, we propose a new disk access interface that allows applications of a storage system (a) to explicitly declare a disk access request as an `update` request to a disk block, in addition to the standard `read` or `write` request and (b) to associate with an update request a callback function that performs the actual update on its target disk block. With disk update requests explicitly labeled, a storage system can now aggregate them, including the implicit reads contained within, in the same way as it handles the disk write requests. With access to application-specific disk block update functions, a storage system can directly apply proper updates to each disk block retrieved on behalf of the processes issuing the disk update requests, thus gaining much more flexibility in disk access scheduling.

The update-aware disk access interface enables a new storage system architecture called *BOSC* (Batching mOdifications with Sequential Commit) [74], which sits between storage applications, e.g., a DBMS process or file system, and hardware storage devices. BOSC is specifically optimized for low-locality update-intensive workloads. In BOSC, incoming disk update requests targeted at a disk block are queued in the in-memory queue associated with the disk block; in the background, BOSC *sequentially* scans the disk(s) to bring in disk blocks whose associated queue is not empty. When a disk block is fetched into memory, BOSC applies all of its pending updates to it in one shot.

BOSC treats each disk update request as an atomic operation and treats them as if they are disk write requests. Under low-locality update-intensive workloads, the resulting throughput improves by an impressive one to two orders of magnitude. In addition, combined with a high throughput low-latency logging technique, BOSC is able to achieve this throughput improvement while

delivering the same durability guarantee and latency as if each disk update request is serviced synchronously.

5.1 Update-Aware Disk Access Interface

The conventional disk access interface provides the following two APIs for applications (including file system and DBMS) to read and write disk blocks, respectively.

- `read(target_block_addr, dest_buf_addr)`
- `write(target_block_addr, src_buf_addr)`

Under this interface, existing storage systems optimize disk write accesses by delaying and/or scheduling them to maximize their throughput and optimizes disk read accesses by servicing them as soon as possible to minimize their latency.

A disk update involves a disk read of the target disk block and then a disk write of the same block after the block is brought into memory and modified. If a storage system could treat each disk update as an atomic operation, theoretically it can delay and schedule the disk reads associated with disk updates in the same way as it does with disk writes. However, to atomically service a disk update request, the storage system must be able to perform the request's intended update operation *without involving the application process issuing the disk update request.* To allow an application running on a storage system to explicitly declare a disk access request as a disk update request and supply the necessary information for the storage system to service it atomically, we propose an *update-aware* disk access primitive specific for disk updates, `modify(target_block_addr, ptr_modification, ptr_commit_function)`, which specifies the target disk block to be modified, a pointer to an application-specific data structure that includes all the information required by the requested modification and a pointer to an application-specific call-back function that commits the actual modification to disk. This primitive is sufficiently general to accommodate disk update requests from such common storage applications as database index managers, including creating a new index entry, updating an existing index entry and deleting an existing index entry.

In the proposed disk update primitive, the application-specific part of each disk update request, i.e., the internal organization of the data structure pointed to by ptr_modification and the internal logic of the function pointed by ptr_commit_function, is fully encapsulated. A storage system implementing the proposed interface carries out the requested modification of each disk

update request by blindly invoking the specified function on the specified data structure, without requiring any knowledge about the data structure and function. Even the developers of BOSC's modify API do not need to know anything about BOSC's internals except following the guideline below when writing a call-back function: "It does not contain any calls to the modify API". In fact, such a storage system does not even need to differentiate among *create*, *update*, or *delete* operations. As a result, the update-aware disk access interface provides the underlying storage system the same flexibility of scheduling disk block create, disk block update and disk block delete requests as disk block write requests and thus the corresponding performance boost.

5.1.1 Caveats with Call-back Function

In the general case, running the call back functions in BOSC entails some security risks like a buggy or malicious program that mismanages system resources. However, these risks are reasonably low for our two target use cases.

1. When BOSC is used as a user-level library that allows a user-level application (e.g. DBMS) to directly manage a disk or disk partition, the background thread that invokes the call-back function runs at the user level and thus can at most compromise the user-level application itself.

2. When BOSC runs as a kernel module, only trusted software components such as file systems are allowed to use the BOSC interface and the call-back functions they registered with BOSC are all kernel code. So, practically speaking, the fact that the background thread invokes these call-back functions in the kernel context does not really introduce any additional safety risks.

If an application calling an update function needs to receive the function's return code in order to proceed, it is NOT appropriate to implement such an update function on top of BOSC, because this use pattern is similar to a read function. In our experiences, there are many applications like garbage collection and continuous data protection systems (CDP), that do not need the called update function to return results.

All the inputs that a call-back function in BOSC needs to run are either logged and put in the per-block queue, or available on the corresponding fetched disk block. At recovery time, the per-block queues at the time of crash are reconstructed and therefore, the call-back function for a given fetched disk block must be able to run after recovery. Here the assumption is that the registered function pointer remains valid across a system crash, which is the case because function pointers are virtual addresses. We also assume that

application binaries are not modified across a system crash and call-backed functions are all statically linked.

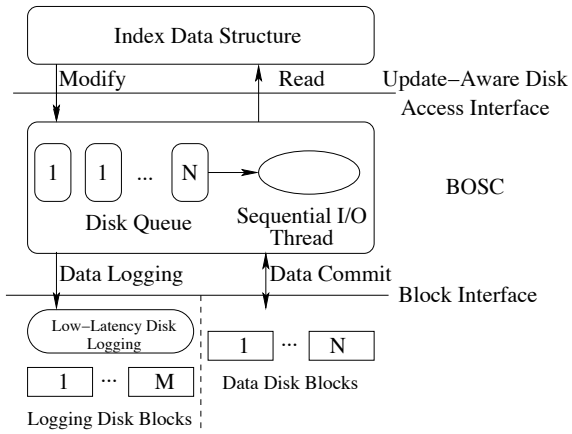

Figure 5.1: *BOSC associates with each disk block an in-memory update request queue. When BOSC receives a disk update request, it logs the request to disk, queues the request in the target block's associated update request queue, and performs the update operation only when the target disk block is brought into memory. BOSC brings disk blocks into memory sequentially according to their logical block addresses.*

5.2 BOSC Architecture

Figure 5.1 shows how BOSC leverages the update-aware disk access interface to aggregate disk update requests and reduce the overall physical disk I/O overhead. BOSC applies a space-efficient low-latency disk logging technique to log each incoming disk update request, queues it in an in-memory per-block request queue, commits updates to disk asynchronously using mostly sequential disk I/O and recovers from failures efficiently.

5.2.1 Low-Latency Disk Logging

Logging a disk update request to disk synchronously and then performing whatever operations triggered by the update asynchronously is a well-known technique. BOSC adopts this technique to service disk updates asynchronously and achieves high sustained disk update throughput, while delivering the same

durability guarantee as synchronous disk updates. Because the end-to-end throughput of BOSC is bounded by its synchronous disk logging performance, BOSC adopts a high throughput low-latency disk logging technique called *Beluga* [186]. The key idea in this low-latency disk logging technique is to write an incoming disk block to where the disk head happens to be rather than moving the disk head to the target location specified by the disk block. More importantly, *Beluga* doesnt predict the disk head position using static disk profiling techniques, rather the data is written continuously so that control over disk head is never lost. Since logging operations need data persistency and do not care about the exact positioning of the data on disk, *Beluga* is given the freedom to choose the target location of the request. Once the data is logged to disk, it's location on disk is updated accordingly in the metadata of the log request. Since *Beluga* requires continuous submission of data to the log disk and if the incoming request rate from BOSC happens to be lower than the logging rate, *Beluga* fills in dummy requests. *Beluga* uses multiple logging disks in case the incoming request rate from BOSC is higher than the logging rate of a single disk. *Beluga* maintains a management header for each log record and it is application-independent. The actual contents of the log records are opaque to *Beluga*. The management header contains the log record size, a valid bit, the associated log device descriptor and a per-device sequence ID. The per-device sequence ID is monotonically non-decreasing and it uniquely identifies the latest committed log record.

An important advantage with *Beluga* is the non-requirement of garbage collection. For example, a 500GB log device could hold the payloads of up to 1 billion 512-byte logging operations without over-writing any old log records, which is large enough to allow *Beluga* to simply wrap around a log device when it reaches the devices end. In case BOSC needs longer retention period for its log records, atleast two disk arrays should be provisioned, so that while *Beluga* is logging on one disk array, one can archive logging records in the other log disk to another medium. *Beluga* is explained in greater detail in chapter 6.

5.2.2 Sequential Commit of Aggregated Updates

In addition to *log disks*, BOSC maintains a set of *data disks* to store application data, and a set of in-memory disk update request arrays, one for each data disk. For each data disk containing N data disk blocks, its corresponding in-memory disk update request array contains N slots and each slot contains a per-block disk update request queue associated with the corresponding data disk block. Upon receiving a disk update request, BOSC first checks if the target disk block is memory-resident; if it is, BOSC performs the update against the block immediately, otherwise BOSC appends the update request to the

per-block disk update request queue associated with the request's target disk block and logs the update request to *Beluga*; finally, BOSC returns control to the caller. Because of BOSC's low-latency disk logging, the perceived delay of each disk update request is relatively small, typically smaller than one msec. For each of its data disks, BOSC maintains two background threads, disk I/O thread and update thread. An update thread is woken up whenever its corresponding disk I/O thread fetches some data to be processed. The update thread applies any pending updates to the buffer fetched by the disk IO thread, by dequeuing and calling the callback_ptr for every request in the corresponding per-block disk update request queue. Upon successfully completing the pending updates, the update thread notifies the corresponding disk I/O thread accordingly. A disk I/O thread constantly sweeps the data disk from the beginning to the end in a sequential manner. During every sweep, if a block's corresponding per-block disk update request queue is non-empty, the disk I/O thread reads that block into a buffer stored in main memory and passes the buffer to its corresponding update thread. Once the update thread notifies its corresponding disk I/O thread of a successful update, the disk I/O thread submits the updated in-memory data block to the data disk. To further minimize the disk access overhead in this read-modify-write loop of commit processing, instead of processing one disk block at a time, the disk I/O thread physically reads and writes a continuous run of disk blocks and commits pending updates on a run-by-run basis.

A disk block run corresponds to a contiguous sequence of disk blocks, such that the number of disk blocks in the sequence is no greater than a threshold (currently set to 32 blocks) and the percentage of non-empty per-block disk update request queues corresponding to the disk blocks in the sequence, is above another threshold (currently set to 0.5). These two thresholds are just heuristics based on our experiences. On order to minimize the synchronization overhead, the disk block runs must not only be disjoint, but even the first and last disk block in it must have a non-empty per-block disk update request queue. As the disk I/O thread reads a disk block run R2 from the data disk, the update thread applies pending updates to another disk block run R1 that has previously been brought into memory from that data disk. By the time the disk I/O thread finishes reading R2, if the update thread has completed applying all the pending updates to R1, then the disk I/O thread proceeds to submit R1 to disk. By pipelining the processing of runs, BOSC is able to completely eliminate any unnecessary disk seek delays, and reduce the number of disk rotations, in processing a run, to two.

5.2.3 Recovery Processing

In case of a system crash, *Beluga* doesn't initiate recovery procedure because *Beluga* is indifferent to the payload of the logging records and hence the onus of handling recovery procedure is on BOSC. After a system crash, when BOSC boots up, it first fetches a chunk of records that were logged the latest. BOSC periodically logs a special metadata log record to indicate the youngest record that's committed to data disk. So BOSC just needs to request *Beluga* to fetch in a chunk of records that were last written to log disk and scan them to discover uncommitted disk update requests. BOSC then reconstructs the in-memory per-block disk update request queues that exist immediately before the crash and resumes its normal processing. This is similar to the idea employed in Aries [105] with some optimizations as described below. Note that BOSC chooses *not* to commit all uncommitted disk update requests to disk at recovery time. Instead, it merely aims to reconstruct the in-memory per-block update request queues and relies on BOSC's normal sequential commit mechanism to write them to disk. More concretely, BOSC's recovery procedure consists of the following steps:

1. Requests *Beluga* to search the log for the log record with the largest sequence ID,

2. Determining the *replay window* in the log that contains log records required for the reconstruction of per-disk-block update request queues and

3. Parsing the log records in the replay window to reconstruct the per-block request queues.

To speed up Step (1), *Beluga* performs a binary search (rather than a sequential scan) of the tracks of the log disk array to track down the youngest log record, which is the last one to be inserted before the crash and thus corresponds to the *end* of the replay window. The binary search works because the disk logging proceeds one track by one track in a FIFO fashion.

Finding the *beginning* of the replay window is trivial because it actually corresponds to the youngest log record, because all update requests associated with log records before the largest sequence ID by definition have already been committed to disk. In Step (3), the log records in the replay window are traversed backwards from the end to the beginning. By exploiting the information in special metadata record, BOSC can avoid inserting a significant percentage of the log records in the replay window into per-block disk update request queues.

5.2.4 Extensions

A straightforward way for a BOSC application like a database index manager to query if a record of a certain qualification exists in a disk block is to explicitly read in the block and scan it for records with the target qualification. However, if the desired record already exists in the disk block's pending update request queue, this approach may bring the target block into memory unnecessarily. To eliminate this unnecessary disk I/O, BOSC provides a query API that allows an application to query a specific disk block: `query(target_block_addr, ptr_query, ptr_query_function)`, where `target_block_addr` is the target disk block's ID, `ptr_query` is a pointer to a data structure containing the query's parameters and `ptr_query_function` is a pointer to an application-specific call-back function that BOSC invokes to search the target disk block's memory-resident update requests queue and/or the target disk block itself if BOSC needs to fetch it into memory. When a disk block is read into memory because it is a target of a read request, BOSC applies all the block's pending updates to it before returning the block to the application issuing the read request. This API allows BOSC to double each in-memory per-block request queue as a cache for the associated disk block.

BOSC treats every disk update request it receives from an application as an independent I/O transaction and is able to guarantee their durability across system failures by synchronous logging and recovery. When a system recovers from a crash, BOSC's recovery manager first restores the side effects of all the disk update requests that BOSC considers already committed and then invokes the application's recovery logic. However, BOSC's I/O transaction is not equivalent to an application-level transaction. If a disk update request U is contained in an application-level transaction that the application's recovery manager thinks should be aborted, it should explicitly issue a compensating update request to undo U. Further, when a transaction calling a BOSC update function is committed after the called BOSC update function is acknowledged, it is OK for the transaction to release all the locks and consider everything is done, even though the actual updates underlying the BOSC update function happen much later. The reason is that BOSC's recovery mechanism guarantees that all acknowledged updates eventually happen.

The current BOSC design is mainly for directly attached disks. To generalize the BOSC idea to a network storage server requires leveraging the security mechanisms developed in the Active Disk [187] project.

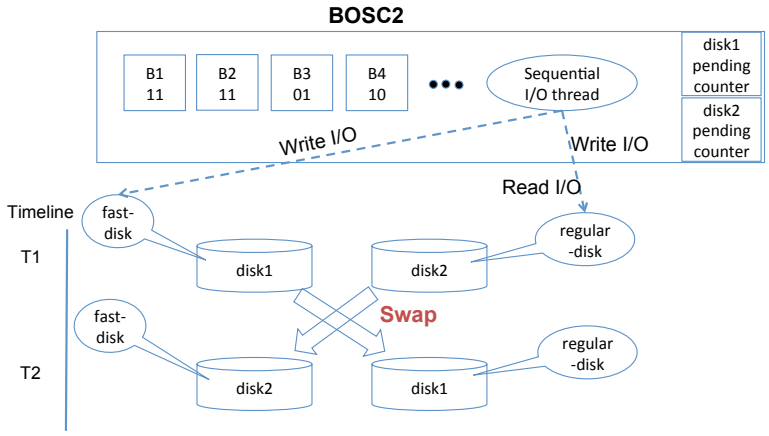

Figure 5.2: *BOSC2 extension with swapping disks, where B1-B4 are in-memory representation of read/write I/O requests and each of these blocks are associated with a 2-bit flag to indicate which of disk1(bit1) or disk2(bit2) has already processed the given I/O request.*

BOSC2: Extending BOSC to Handle Low-Locality Read Workload

When a workload is dominated with low locality, read and update disk requests, the overall performance of any traditional storage system will definitely deteriorate. Since standalone read disk requests typically need to be synchronously serviced, it appears that there isn't much one can do to efficiently handle such a workload. Though BOSC doesn't deteriorate the performance of standalone read disk requests (shown later in the evaluation section), it neither improves its performance, because frequent standalone random read disk requests prohibit the sequential sweeps of the background disk I/O thread. However, the BOSC technique could be adopted in a new storage system where the update and write I/O requests are handled differently from the standalone read disk requests, and such a system could possibly avoid the ill-effects of the standalone read disk requests.

We propose a new BOSC-based storage system, BOSC2, which in its simplest setup, uses two data disks, disk1 and disk2, as shown in Figure 5.2. BOSC2 marks one of the data disks as fast-disk and the other data disk as regular-disk. BOSC2 submits the read disk requests to only the disk marked as regular-disk, and submits the write and update disk requests to both the data disks. Since BOSC technique delivers a very high performance in the absence of read disk requests, the fast-disk is expected to deliver a faster throughput than the regular-disk. Since only the write and update disk requests modify the content on the data disks, BOSC2 intends to keep disk1 and disk2 with identical data at all times. When a workload consists of large number of low

locality read disk requests, the difference in throughputs between the two data disks results in a temporary mismatch between the contents of the two data disks, and when the amount of mismatch is beyond a threshold, the markings on the data disks are swapped.

In order to effectively implement the above mentioned proposal, BOSC2 uses the same techniques as BOSC except for a few changes described below. BOSC2 uses the same in-memory queues as BOSC's and it is common to both the data disks. Additionally, BOSC2 uses a counter for each data disk to indicate the number of outstanding I/Os on that disk. BOSC2 also modifies the metadata for each request in the per-block disk update request queue to have an additional field for reference count of one byte. The reference count is initially set to 0 to indicate that it is not committed on either of the disks. The first bit in reference count is set if the request is committed to disk1 and the second bit in reference count is set if the request is committed to disk2. Only when both the first and second bits in reference count are set, BOSC2 removes that request from the in-memory queue. Note that disk1 and disk2 are permanently marked and only the fast-disk and regular-disk markings are temporary. BOSC2 swaps the fast-disk and regular-disk markings on the data disks, when the number of outstanding I/O requests on the regular-disk exceeds beyond a threshold. This threshold factor depends upon multiple factors like the size of the in-memory queue, the raw transfer throughput of the data disk and the percentage of read requests in the input workload. By swapping the markings, BOSC2 allows the previously marked slow throughput regular-disk to now exclude the read disk requests and hence clears its outstanding I/Os in the in-memory queue at a faster rate. On rare occasions, if the percentage of low-locality read disk requests in the input workload is too high, swapping operation happens too frequently and the overall performance of BOSC2 throttles down to an extremely slow throughput which would be similar to what we observe on a traditional storage system.

Swapping operation thus guarantees equilibrium between the disks, thereby ensuring bounded memory requirements for in-memory queue under the assumption that the percentage of read disk requests in the input workload are below a certain threshold. The requirement for two disk I/Os in BOSC2 against one disk I/O in BOSC, shouldn't slow down the overall performance by a large margin because disk I/Os on both the data disks happen in parallel. BOSC2 services the read disk requests either by looking up the in-memory queues or by fetching from the regular-disk. Upon swap operation, the newly marked regular-disk will have the required data either on its disk or in the in-memory queue waiting to be committed to the disk. Either ways, BOSC2 returns the latest available data for every read disk request. Thus the proposed

technique ensures that BOSC2 delivers a consistent high throughput that is largely unaffected by the presence of read, write or update disk requests in the input workload.

5.3 Applications of BOSC

To demonstrate usefulness of BOSC across wide range of storage systems, BOSC is applied to B+ tree and hash table because these two are amongst the most used data structures in many storage systems. We also demonstrate the application of BOSC in two real world applications:

Metadata management in a deduplication system's garbage collector: Garbage collection system in DSS shows orders of magnitude improvement when BOSC mechanism is applied to handle update I/O requests in its reference count array. These along with more details were discussed in detail in Chapter 3.

Index management in a continuous data protection (CDP) system: We built a CDP system called *Mariner* [188] that logs every block-level disk write request on a storage system to a protected storage server and creates a new version for a logical disk block whenever it is over-written. *Mariner* maintains a map between each logical block number and its physical block numbers, each corresponding to a distinct version. Upon receiving a block-level disk write request, *Mariner* logs the write request's payload to create a new version and inserts a new entry into the map. For larger indices, most of the map lookup and update operations require disk IO access, and since most of these IO operations result in random disk IOs, BOSC technique is applied to extract maximum performance from Mariner. DISCO adopts Mariner to build its snapshot table (discussed in Section 1.2.3), but since a CDP is a generic technique referred in the literature, we will address a generic CDP system in this work. Section 5.4.5 gives a detailed overview of the performance evaluation on Mariner.

5.3.1 BOSC-Based B^+ Tree

We have successfully ported B^+ tree index implementation from TPIE [189, 190] to the BOSC storage system prototype. TPIE is a software environment written in C++ that is designed specifically to minimize the disk I/O cost in very large data sets.

The BOSC-based B^+ tree assumes all internal tree nodes and a small subset of leaf nodes are memory-resident. To service a modification query that inserts, deletes, or updates an index record, the BOSC-based B^+ tree first traverses the internal nodes to identify the leaf node containing the target index record, then constructs a disk update request record and finally calls BOSC's disk update API using the target leaf node's disk block address, the associated update request record and the corresponding commit function as input arguments. Upon receiving such a disk update request, BOSC logs the request to the log disks first, commits the update to the target leaf node immediately if it is currently cached in memory, and queues the update request record in the corresponding in-memory request queue associated with the target leaf node otherwise.

To ensure atomicity, the BOSC-based B^+ tree acquires a lock on a leaf node before modifying it and releases the lock after BOSC logs the associated disk update request and queues it in the associated request queue. It is safe to release the lock associated with the target leaf node of a modification query before physically committing the requested modification to disk, because BOSC *guarantees* the effects of a modification query's associated disk update request be visible to all subsequent queries that access the same leaf node, even in the presence of power failures.

An implicit assumption underlying the design of BOSC is that each disk update request modifies only its target disk block. However, this assumption does not always hold for B^+ tree, because a modification to a tree node, e.g., an insertion of a new index record, may trigger a restructuring of the tree and thus modifications to other tree nodes. If a disk update request that triggers additional disk updates is not processed immediately at the time when it is queued but deferred until the time when it is committed to disk, a disk block's in-memory request queue may grow unbounded, because the triggered restructuring may be recursive. This makes the update commit processing time of a disk block less predictable and increases the response time of read query requests because servicing read query requests requires scanning of per-block update request queues.

To mitigate the performance overhead due to disk update requests that trigger additional disk updates, the BOSC-based B^+ tree maintains a count for the *effective* number of index records in each leaf node, including the pending delete and insert operations and *proactively* triggers the split of a leaf or internal node when the effective number of records in a tree node exceeds a threshold, say 70%. If the leaf node to be split does not have any index records on disk, all the node's index records are in the associated update request queue and the BOSC-based B^+ tree performs the split without incurring any disk

accesses. If the leaf node to be split has some index records on disk, the BOSC-based B^+ tree defers the split operation until the time when these records are brought into memory by the background BOSC thread.

5.3.2 Hash Table

We implemented two persistent Hash Table implementations: One is the vanilla implementation based on the conventional disk read/write interface and the other is built on top of BOSC. The vanilla implementation is a traditional disk based hash table implementation where for every hash key, the corresponding bucket is a file stored on disk. For every disk write request corresponding bucket is fetched into memory and updated with the given record and then written back to disk. The BOSC based implementation synchronously buffers the disk write requests in an in-memory queue along with logging to disk and a separate background thread sequentially handles each bucket file on disk. The naive callback function simply writes the record to bucket without any fuss.

5.4 Performance Evaluation

5.4.1 Evaluation Methodology

We have built a complete Linux-based BOSC prototype. This prototype supports the update-aware disk access interface as well as sequential commit of aggregated disk updates. On top of this BOSC prototype, we built a BOSC-based B^+ tree implementation, which is derived from TPIE and took about 2 man weeks.

To evaluate the efficiency of the BOSC-based B^+ tree, we used the following two synthetic workloads: (1) *random insert* workload, (2) *random update* workload. In the random insert workload, records with randomly generated key values, which fall between 0 and the pre-defined index size, are inserted into an initialized index. The random update workload updates random existing records that are inserted by the random insert workload.

The evaluation testbed for the BOSC prototype is a Dell PowerEdge 600SC machine with an Intel 2.4GHz CPU, 512KB L2 cache, 4GB memory, a 400MHz front-side bus, two Gigabit Ethernet interfaces and five 7200-RPM IBM Deskstar DTLA-307030 disks, four of which store the B^+ tree index records and one of which is dedicated to low-latency logging.

To be realistic, the initial B^+ tree must contain a substantial number of index records, on the order of tens of gigabytes of data. If we were to measure

the throughput of the BOSC-based B^+ tree implementation against an initially empty B+ tree, then the measurement results for initial inserts/updates would be biased as they don't include such critical components as lock acquisition and tree traversal. However, it takes several hours to generate a properly initialized multi-gigabyte B^+ tree and we need dozens of initialized B^+ trees, each with a different node or record size, in the entire evaluation study. So a fast B^+ tree initialization method is needed. The major bottleneck in the B^+ tree initialization process is the disk I/Os required to put leaf node data on disk. Because the actual contents of the leaf nodes are immaterial to our evaluation study, we could completely skip these disk I/Os in the initialization process and focus only on the creation of internal tree nodes. Therefore, when a B^+ tree is initialized this way, only its internal nodes are properly set up and its leaf nodes are only allocated on disk but not actually initialized. During the experiment run, whenever a leaf node is brought into memory for the first time, its content is filled with proper values at that point. The values filled are calculated on the fly, because the structure of the initialized B^+ tree and the key values in it are pre-determined. This B^+ tree initialization method proves invaluable to our evaluation study, because it saves us hundreds of hours, e.g., the time to initialize a 64-Gbyte B^+ tree is reduced from 36 hours to under 50 seconds.

5.4.2 Logging disk, Data disk combinations

We measured both the insert throughput and read throughput to get an overall measure of BOSC performance. We see that the performance is maximum for the case of 1 Logging disk and 4 Data disks as shown in figure 5.3.

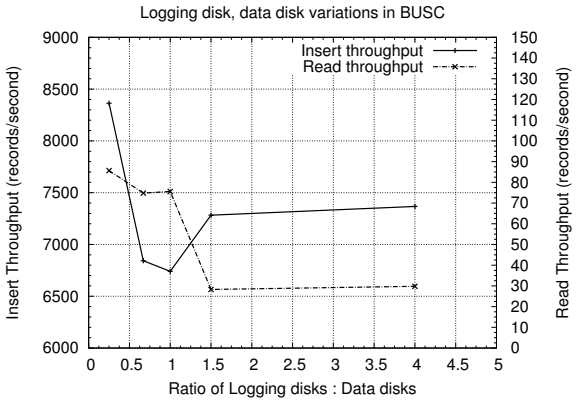

Figure 5.3: Insertion and read throughput for various logging disk/data disk variations

With the increase in data disks, insert throughput increases because when we have more data disks we can do more data splicing and when I/O thread commits records to disk, disk I/O happens on multiple data disks in parallel. In case of adding more logging disks, we can bring more parallelism and increase the logging throughput. But we already have a very fast logging operation that can log with a latency of under 1 ms. The main bottleneck in BOSC is I/O, which commits records to disk. Hence with a fixed amount of resources, it's preferable to dedicate more disks to data disks than to logging disks. But on comparison between 2 logging disks, 3 data disks vs 3 logging disks, 2 data disks: We see that latter has a better throughput in spite of lower number of data disks. This is an anomaly, because the low read throughput gives more time for IO thread to commit more records to disk. Hence by the time read thread finishes and insertion thread starts, the per-block queue is almost empty and hence the insertion thread doesn't block for want of memory and hence insert throughput looks to be high.

We also see that read throughput drops with the decrease in data disks. There is only a slight drop in read performance and this is because of the drop in cache hit percentage. We measure the cache hit percentage as the number of times we find a record in per-block-queue in memory out of total number of read requests issued. It is not straightforward to reason out the read performance or drop in cache hits, because we cannot predict when the I/O thread commits the records in per block queue. TPCC trace might want to fetch some records and the I/O thread could have already committed that record to disk. So this can happen even if the I/O thread is slow or fast. But overall, we can guarantee that BOSC read performance does not deteriorate very much compared to vanilla $B+$ tree. For the last 2 cases, read throughput is very low because number of data disks are very low. When number of data disks are very low, insertion thread operates very slowly and hence the entire $B+$ tree operation proceeds very slowly. It's observed that cache hit percentage is 0.11% when data disks are too low. The ratio is around 0.4% for the other cases. In another set of experiments, we will see how these cache hit percentages keep changing with memory size. In a normal case, read throughput should be around 50 records/second, but in the case of more log disks and less data disks, we see that read throughput decreases to as low as 28-30 records/second. That's because, cache hit ratio is very bad and also read thread is in contention with IO thread, which is processing records very slowly. Note that IO thread and read thread will frequently run in contention for global $B+$ tree lock and hence lock contention increases a lot in the case of slow read thread processing.

5.4.3 Overall Performance Improvement on B^+ Tree

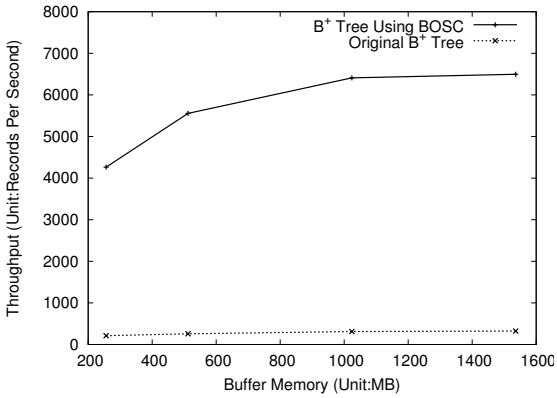

Figure 5.4: BOSC vs. Vanilla for Random *insert* workload. Comparison between the record insert throughput of a BOSC-based B^+ tree implementation and a vanilla B^+ tree implementation based on the conventional disk read/write interface under the random insert workload, when the total amount of buffer memory is varied from 256 MB to 1.5 GB. The leaf block size is 64 KB, the record size is 64 B and the initial index size is 16 GB.

Figures 5.4 & 5.5 show the throughputs of a vanilla B^+ tree implementation on a conventional disk read/write interface (with 5 data disks) and a BOSC-based B^+ tree implementation under the random insert and random update workload respectively. The throughput of the vanilla B^+ tree implementation increases only slightly with the buffer memory because the poor locality in the random insert workload does not offer much room for leaf node caching to be effective. In contrast, the throughput of the B^+ tree implementation keeps improving with the increase in buffer memory size because more pending insertion requests can be accumulated in each sequential commit cycle. This improvement saturates at 1024 MB because the given buffer memory exceeds the product of the new record insertion rate and the sequential commit cycle length. When the buffer memory size is 1024 MB, the sustained throughput of the BOSC-based B^+ tree implementation under the random insert workload reaches around 6410 requests/second, which is *20 times* higher than that of the vanilla B^+ tree implementation using the conventional disk read/write interface (311 records/second). When buffer memory is not the performance bottleneck, the throughput of the BOSC-based B^+ tree implementation is mainly bound by the physical disk I/O efficiency in the sequential commit process.

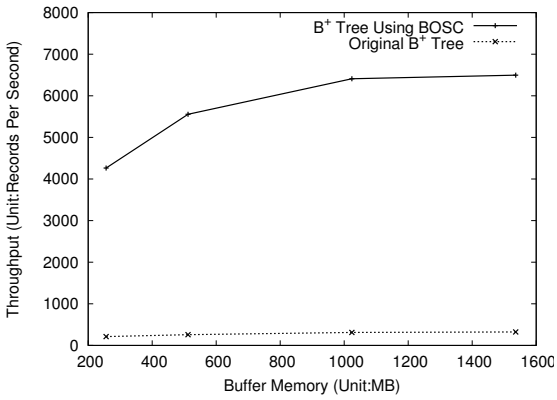

Figure 5.5: BOSC vs. Vanilla for Random *update* workload. Comparison between the record update throughput of a BOSC-based B^+ tree implementation and a vanilla B^+ tree implementation based on the conventional disk read/write interface under the random update workload, when the total amount of buffer memory is varied from 256 MB to 1.5 GB. The leaf block size is 64 KB, the record size is 64 B and the initial index size is 16 GB.

The performance of the BOSC-based B^+ tree implementation under the random update workload is almost the same as that under the random insert workload, because in both workloads the accesses to the index pages are random and consequently their performance is bottlenecked by disk I/O. Figure 5.5 shows that the throughput improvement of the BOSC-based B^+ tree implementation over the vanilla B^+ tree implementation is the same as in Figure 5.4 These two results conclusively demonstrates BOSC is as efficient for an *update-in-place* workload as for an *insert-only* workload. In contrast, most previous B^+ tree optimizations [90, 93, 96] are only applicable to insert-only workloads.

The two key performance-boosting features of BOSC are low-latency logging and asynchronous sequential commit using multiple request queues. A simpler alternative to BOSC's low-latency logging is logging by appending to the end of a file. A simpler alternative to asynchronous sequential commit is to queue all update requests in a single queue and batch-commit the head N requests in the queue according to their target disk block addresses. To evaluate the performance contribution of each of these two features, we compare the throughputs of the following four B^+ tree variants. The first variant, called *One-Queue-Append*, appends each incoming update request to the end of a log file and inserts it into a single FIFO queue. The second variant, called *One-Queue-Trail*, uses low-latency logging to log each incoming update request and inserts it into a single FIFO queue. The third variant, called *Multi-Queue-*

Append, appends each incoming update request to the end of a log file and inserts it into the per-block queue associated with its target block. The fourth variant is BOSC, which uses low-latency logging to log each incoming update request and inserts it into the per-block queue associated with its target block.

To demonstrate the performance benefits of BOSC under more realistic workloads, we collected a trace of access requests to the index engine of the MySQL DBMS under the TPC-C workload [191], where the number of warehouses is set to 20, 40, 60 and 80. Each trace entry includes the type (e.g. read, update, delete and insert) and the key/data information of each request issued to the index engine. For each warehouse number, we ran the TPC-C workload for three hours to generate an index of the size 16 GB, 32 GB, 48 GB and 64 GB, respectively and collected the corresponding access request trace. For each index access trace collected, we replayed the first half to create an initial image of the database index and then replayed the second half and measured the throughput of the input requests in the second half of the trace.

Figure 5.6 compares the throughputs of these four B^+ tree implementation variants under four different TPC-C traces. Across all warehouse parameters, as expected the BOSC-based B^+ tree implementation tops the four variants with the best throughput. For example, when the warehouse number is 80, the throughput of the BOSC-based B^+ tree is 6058 requests/second, as compared to 20 requests/second for the *One-Queue-Trail* scheme and 2386 requests/second for the *Multi-Queue-Append* scheme. The performance gain of BOSC over the *Multi-Queue-Append* scheme comes from low-latency logging, which maximizes logging efficiency and thus the overall update throughput. The fact that the BOSC-based B^+ tree implementation is more than 2.5 times faster than the *Multi-Queue-Append* variant (the Y axis is in log scale) shows the importance of lower logging latency. The BOSC-based B^+ tree implementation is more than 300 times faster than the *One-Queue-Trail* variant, which shows the importance of sequential commit as enabled by multiple request queues is much more than low-latency logging. There is no noticeable performance difference between the *One-Queue-Trail* variant and the *One-Queue-Append* variant because both are bottlenecked by the excessive disk access overhead associated with committing pending updates to disk.

Larger warehouse number corresponds to larger database index size and lower access locality. The fact that the throughput of the BOSC-based B^+ tree implementation remains largely independent of the warehouse number suggests that BOSC enables a B^+ tree implementation to exhibit good throughput without relying on the input workload's locality characteristics. Overall, under the TPC-C workload, the BOSC-based B^+ tree implementation is 300 times

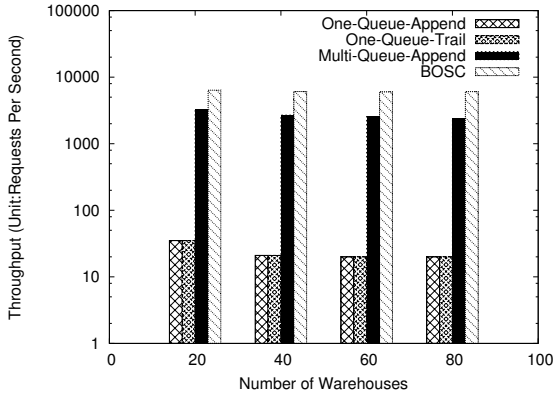

Figure 5.6: *Throughput comparison among the BOSC-based B^+ tree implementation, the B^+ tree implementation with multiple request queues and append-only logging, the B^+ tree implementation with one request queue and append-only logging and the B^+ tree implementation with one request queue and low-latency logging under the four index access traces collected by running the TPC-C workload with different warehouse numbers against MySQL. The Y axis is in log scale. The leaf node size is 64 KB and the buffer memory is 1 GB.*

faster than that of the vanilla B^+ tree implementation when there are 80 warehouses and is 180 times faster when there are 20 warehouses.

Sensitivity Study

In this section, we evaluate the impacts of the leaf node size, the index record size, the total index size and the buffer memory size on the performance of the BOSC-based B^+ tree implementation. Each experiment run starts with a fixed-sized initial B^+ tree and continues with index record insertions/updates until the first *sequential commit cycle* is completed. At that point, we measured the total number of insertions/updates and the elapsed time.

In evaluating the impact of different parameters on the insert/update rate, there are 3 factors to consider: (1) the disk I/O efficiency, which reflects how effectively the background commit thread removes unnecessary disk access overhead, (2) the degree of batching, which determines how many requests over which each disk I/O operation's cost is amortized, and (3) the CPU overhead associated with traversing from the B^+ tree's root to the target leaf node of a given insert/update request and queuing pending requests.

The throughputs of the BOSC-based B^+ tree under the random insertion, sequential insertion and clustered insertion workload when the leaf node size varies are shown in Figure 5.7. The throughput performance of the BOSC-based B^+ tree index under the sequential insert workload is much higher than

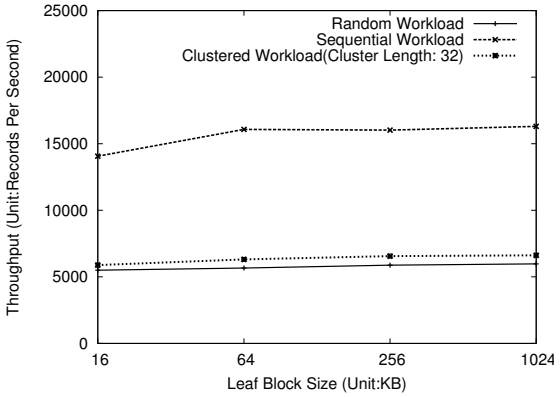

Figure 5.7: *Effect of leaf node size variation on the throughput of a BOSC-based B^+ tree implementation under the sequential insertion, clustered insertion and random insertion workload. Leaf node size is varied from 16 KB to 1024 KB. The X axis is in log-scale. The Y axis is the number of new records inserted per second. The memory allocated for all per-block request queues is 1 GB, the record size is 64 B and the initial index size is 64GB.*

that under the random insert workload for two reasons. First, the average number of pending requests in each queue at the time of commit is higher under the sequential insert workload than that under the random insert workload. Second, the CPU overhead of processing insert/update requests is lower under the sequential insert workload than that under the random insert workload because of fewer L2 cache misses. For the random insert workload, it takes around 140 micro-seconds to complete an insertion request, whereas it takes only 67 micro-seconds for the sequential insert workload.

As the size of the test B^+ tree index's leaf block is increased, more index records can be packed into each leaf block, the degree of batching in terms of number of pending requests per disk block fetched is increased and so is the throughput of the BOSC-based B^+ tree index, as shown in Figure 5.7. This effect is more pronounced under the clustered and sequential insert workload than under the random insert workload, because there is not much batching in the random insert workload anyway.

Figure 5.8 shows that as the size of the test B^+ tree's index record is increased, fewer index records can fit within each leaf block and the degree of batching in terms of number of pending requests per disk block is decreased. The throughput degradation for the clustered insert and random insert workload is directly correlated with the decrease in the degree of batching, but that for the sequential insert workload is mainly due to additional L2 cache misses during insert request processing.

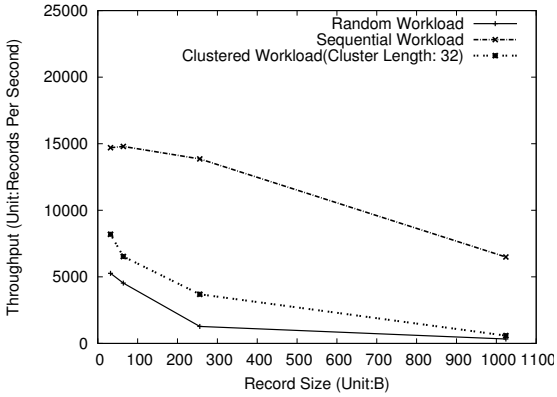

Figure 5.8: *Effect of index record size variation on the throughput of a BOSC-based B^+ tree implementation under the sequential insertion, clustered insertion and random insertion workload. Index record size is varied from 64 B to 1024 B.*

If both leaf block size and index record size are increased while keeping their ratio constant, the number of index records per leaf block remains the same, but the degree of batching in terms of number of pending requests per fixed-sized disk I/O is still decreased, e.g., the effective number of pending requests committed per 100-KB disk I/O decreases as the leaf block size is increased from 8KB to 64KB, and so is the throughput of the BOSC-based B^+ tree index, as shown in Figure 5.9.

As the total B^+ tree index size is increased, the average number of pending requests accumulated in each per-block queue within one sequential commit cycle becomes smaller, the degree of batching at the time of commit is thus decreased and so is the throughput of the BOSC-based B^+ tree index, as shown in Figure 5.10. The throughput impact of the index size is less obvious under the sequential insert workload because the degree of batching remains largely constant regardless of the index size. Figure 5.10 also shows that the performance of the BOSC-based B^+ tree implementation under the random update workload is almost the same as that under the random insert workload, because in both cases accesses to the index pages are random and consequently their performance is bottlenecked by disk I/O.

As the buffer memory for per-block request queues is increased, the number of pending requests at the time of commit is increased, the degree of batching is increased and the overall throughput under the random insert and clustered insert workload are increased as shown in Figure 5.11. The performance impact of buffer memory size is minimal for the sequential insert workload because its degree of batching is already quite high and largely unaffected by the buffer memory size.

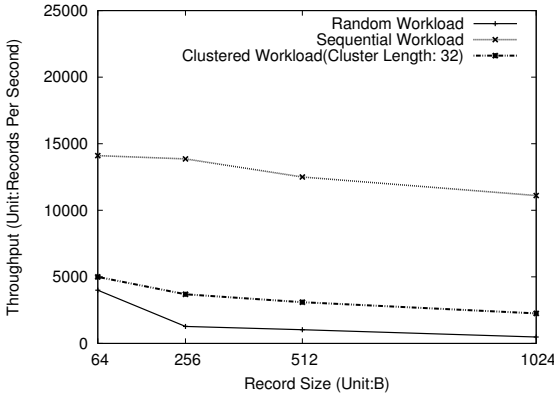

Figure 5.9: *Effect of varying both the index record size and leaf block size on the throughput of a BOSC-based B^+ tree implementation under the sequential insertion, clustered insertion and random insertion workload. The index record size is varied from 64 B to 1 KB and the leaf node size also varies proportionally so that the ratio between the two is* fixed.

Read Query Latency

Although BOSC is designed to optimize the throughput of low-locality update-intensive workloads, it does *not* degrade the latency of read accesses to database indexes built on top it. This is unusual, because many previously proposed B^+ tree implementations optimized for the same workload tend to trade better update throughput for longer read latency.

Index Structure	**Point Query (Unit: msec)**		**Range Query (Unit: msec)**	
	With BOSC	**Without BOSC**	**With BOSC**	**Without BOSC**
B^+ Tree	10.20	10.19	15.75	15.76

Table 5.1: *The average latency of Point and Range queries for the B^+ tree implementations with and without BOSC. The leaf block size for all index structures is 4 KB and the buffer memory is 256 MB.*

Table 5.1 shows the average latency of 100 Point and Range queries against a pre-populated B^+ tree using the B^+ tree implementations with and without BOSC. For *point* queries, the key values are generated randomly from the underlying key space. For *range* queries, the starting key values are generated randomly from the key space and the maximum range size is fixed at 1,000.

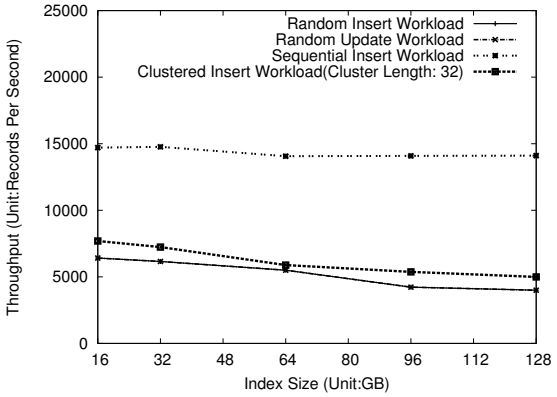

Figure 5.10: *Effect of index size variation on the throughput of a BOSC-based B^+ tree implementation under the sequential insertion, clustered insertion, random insertion and random update workload. The initial index size is varied from 16 GB to 128 GB.*

There is no statistically significant difference between the average read query latency of the BOSC-based B^+ tree implementation and that of the vanilla B^+ tree implementation, even though the read-path processing in BOSC requires an additional step of searching the target block's in-memory request queue. BOSC's caching has little effect on it's read performance because the locality in the input workload is relatively low, as evidenced by the relatively large average latency. This result demonstrates that the update/insert performance gain of BOSC does not come at the expense of read performance degradation, which is often the case for other B^+ tree optimizations [90, 93, 96].

5.4.4 Overall performance improvement on Hash Table

We applied the same random insert and update workload used in the evaluation of B^+ tree implementations to evaluate two persistent Hash Table implementations: One is the vanilla implementation based on the conventional disk read/write interface and the other is built on top of BOSC. Each index record inserted is 16 bytes long and includes an 8-byte key. The hash table used in this experiment occupies a 20-GB disk partition and is initialized with a *sequential insert workload* whose key value starts with 0 and is increased with an increment of 1,000,000 until 10 Gbytes worth of records are inserted. Each experiment run consists of insertions of new records into an empty hash table until 8 Gbytes worth of new records are inserted.

Given a fixed amount of buffer memory, we used the memory to cache the hash table's data pages in the case of the vanilla hash table implementation

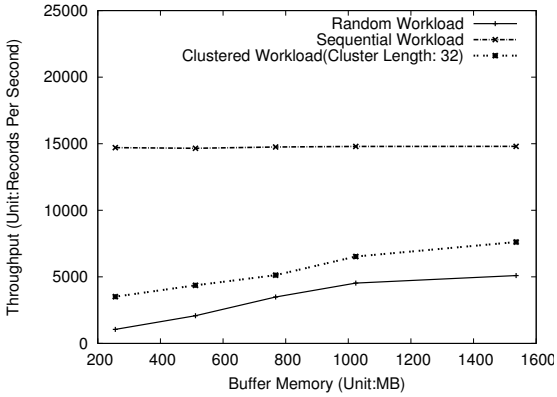

Figure 5.11: *Effect of buffer memory size variation on the throughput of a BOSC-based B^+ tree implementation under the sequential insertion, clustered insertion and random insertion workload. The BOSC's buffer memory is varied from 512 MB to 1536 MB.*

and to hold per-block request queues in the case of the BOSC-based hash table implementation. We used the random insert workload and set the scan size to 256 KB. Figure 5.12 shows that the throughput of the vanilla hash table implementation increases slightly with the buffer memory size because larger buffer memory size improves the buffer cache hit ratio of disk accesses. In contrast, the throughput of the BOSC-based hash table implementation with a disk I/O unit size of 256 KB improves dramatically with the increase in the buffer memory size until 960 MB, at which point the number of pending insertion requests it can batch per disk I/O unit levels off.

When the buffer memory size is smaller than 64 MB, the average queue length of the BOSC-based hash table implementation is 1 and so the performance gain of BOSC originates mainly from sequential disk I/O. When the buffer memory size is 960 MB, the throughput of the BOSC-based hash table implementation under the random insert workload reaches around 23006 requests/second, which is more than 50 times higher than the vanilla hash table implementation (445 records/second). Under the random insert workload, when the buffer memory size is 960 MB, the average amount of time required to read and write a disk I/O unit is 16.4 msec and the number of insertion requests committed per 4-KB page is 6, therefore the update throughput should be $\frac{6*256KB/4KB}{16.4 \cdot 10^{-3} second} = 23414$ requests/second, which approximately matches the empirical throughput measurement.

Figure 5.13 shows the throughput improvement of the BOSC-based hash table implementation over the vanilla hash table implementation under the random update workload is identical to that under the random insert work-

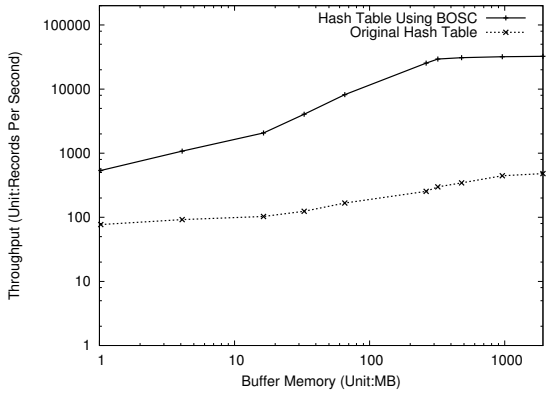

Figure 5.12: *Comparison between the record insertion throughput of a BOSC-based Hash Table implementation and a vanilla Hash Table implementation based on the conventional disk read/write interface under the random query workload when the total amount of buffer memory is varied from 1 MB to 2 GB. Both X and Y axes are log-scale. The scan size is 256 KB.*

load. This once again demonstrates that BOSC is as effective in improving the performance of update-in-place workloads as in improving the performance of insert-only workloads.

5.4.5 Application of BOSC to *Mariner*

To measure the overall disk write logging throughput of *Mariner*, we ran a kernel thread inside *Mariner* that constantly generates new 4KB block versions, with their logical block numbers uniformly and randomly distributed in $[0, 2^{41}]$. During the experiment run, there are totally 20 GB worth of new block versions logged and 935 MB ($20GB * \frac{24}{512}$) worth of indexed map entries inserted. The buffer memory for BOSC's per-block update request queuing is set to 64 MB and the leaf page cache for TPIE is also set to 64 MB.

The disk write logging performance of *Mariner* when the indexed map is not updated at all is mainly determined by the overhead of logging new block versions and thus represents its performance upper bound.

As shown in Figure 5.14, compared with the upper bound, the throughput degradation due to the indexed map update is up to 95% when the indexed map is implemented as a vanilla B^+ tree (i.e. TPIE) is up to 80% when the indexed map is implemented as a B^+ tree using file-level append-only logging and per-block request queuing and is no more than 15% when the indexed map is implemented as a BOSC-based B^+ tree. This result demonstrates that the BOSC-based B^+ tree implementation successfully removes indexed map

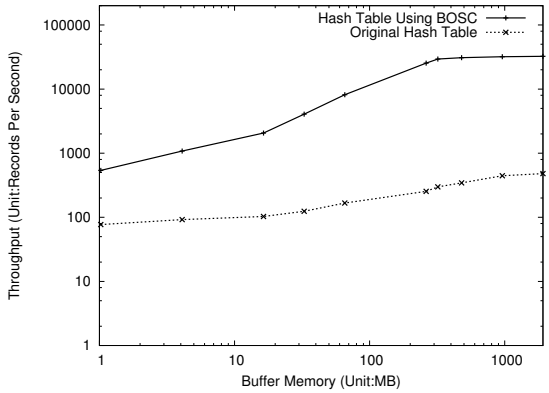

Figure 5.13: *Comparison between the record update throughput of a BOSC-based Hash Table implementation and a vanilla Hash Table implementation based on the conventional disk read/write interface under the random query workload when the total amount of buffer memory is varied from 1 MB to 2 GB. Both X and Y axes are log-scale. The scan size is 256 KB. Note that unlikes Figure 5.12, the input workload consists of update operations.*

update as a performance bottleneck of *Mariner*. Across these three variants the throughput degradation decreases with the increase in the inter-request interval, because the additional indexed map update overhead becomes less significant when the input load is less demanding. In terms of absolute performance, the disk write logging throughput of the *Mariner* version using a BOSC-based B^+ tree is more than 45 times higher than that of the *Mariner* version using a TPIE-based B^+ tree and is more than 3 times better than that of the *Mariner* version using a BOSC-based B^+ tree without low-latency logging.

5.5 Summary

Update-itensive random disk I/O workloads are very common in a storage system like DISCO. With a conventional disk access interface, the performance of the entire storage system drops down drastically. Currently no good solution can effectively handle such workloads without resorting to special caching hardware such as battery-backed DRAM. We propose a novel disk access interface called BOSC, that describes a simple but effective solution to the random update disk I/O problem. BOSC provides a new disk access interface to the storage system, and an efficient batched processing strategy that converts the random updates to mostly sequential disk I/Os.

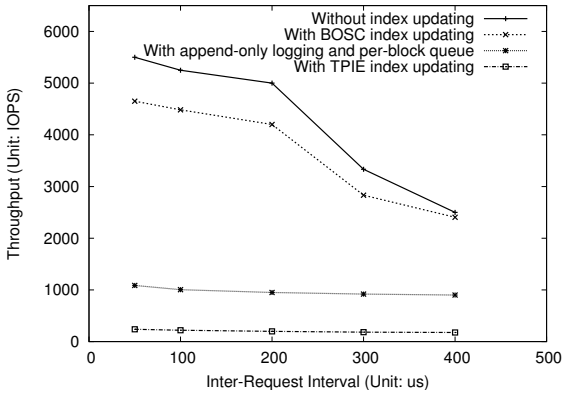

Figure 5.14: *The block-level logging throughputs of four* Mariner *variants: The first variant does not update any index; the second variant updates the map index using a BOSC-based B^+ tree; the third variant updates the map index using a B^+ tree that uses append-only logging and multiple per-block queues; The fourth variant updates the map index using a vanilla B^+ tree.*

We have successfully built a BOSC prototype that embodies these two ideas and demonstrated how BOSC eliminates the disk bottleneck issue in *Sungem*'s garbage collector and improves the overall performance of DISCO's backup operation. We also empirically demonstrated the efficiency of BOSC by showing that the update request throughput of BOSC-based B^+ tree and hash table implementations are more than an order of magnitude faster than that of their vanilla versions built on top of the conventional disk access interface. In summary, the specific research contributions of this work include:

- A new disk access interface that supports disk update as a first-class primitive and enables the specification of application-specific callback functions to be invoked by the underlying storage system,
- A highly efficient storage system architecture that effectively commits pending update requests in a batched fashion and drastically improves the physical disk access efficiency by using mostly *sequential* disk I/O to bring in the requests' target disk blocks, while ensuring the same durability guarantee as servicing the update requests synchronously.

Chapter 6

High Throughput Low Latency Disk Logging

A well-known technique to enhance the throughput of write-intensive disk-based data processing systems is to synchronously log the incoming updates to disk and asynchronously commit them to their corresponding disk locations in a way that is optimized for the commit performance. We saw in Chapter 5 how the BOSC-based storage system is designed to efficiently adopt such a technique. The user-perceived response time of an update request to such systems is determined by the latency of the request's associated logging operation, because the update request is generally considered done when its logging step is completed.

An ideal disk logging system is one that provides high throughput and low latency for logging operations with small payloads, e.g. 64 or 128 bytes. Small-payload logging operations are important because many applications only need to log the information associated with high-level operations, such as an update to a record in a B-tree page or a hash table bucket, and the size of this information is typically small. Low logging latency is also crucial, because it directly impacts the user-perceived response time and because many applications are designed to be latency-bound, i.e., they cannot produce more requests unless previously submitted requests are completed. Such applications are very common in online transaction processing (OLTP) systems. We even saw in Chapter 5, with examples of various applications that exhibit such a workload, and all these applications require a fast logging system. There are several components in DISCO, that benefits the most with such a fast disk logging system. DMS maintains a set of disk-based mapping tables to translate addresses between the three addressing namespaces in DISCO (discussed in Section 1.2.1) and the size of each entry is typically less than 128 bytes. The requests in the input workload to these mapping tables have extremely

low locality and require very low latency processing because these requests correspond to real-time disk access requests on the VDs. Therefore, in order to ensure good overall performance of the SDDS system, the DMS has to process such a low locality workload with high throughput and low latency, which in turn requires the disk logging system to address these needs. As another example, the P-array in garbage collection (GC) component of *Sungem* (discussed in Section 3.3.2) maintains a large disk-based metadata index to store metadata for each physical block in the entire storage system. Once again the input workload has extremely low locality and is update-intensive, and hence GC uses BOSC technique to process such a workload, which in-turn requires a high throughput low latency disk logging system.

Because logging involves append operations and thus sequential accesses, the conventional wisdom is that optimizing the performance of logging operations is relatively straightforward. However, our research suggests that it is actually not at all trivial to achieve both high throughput and low latency for logging operations, especially for fine-grained ones and has identified three key challenges. First, there is a mismatch between fine-grained logging and modern file systems. More concretely, because most modern file systems use 4KB block as the basic unit of reading and writing, logging a 64-byte or 128-byte record to a log file may require a read of the log file's last block and a write of the same block after appending the log record to it. Second, there are multiple steps on the data path from the system call interface to the disk platter that a logging operation's payload needs to traverse, and some of these steps incur a per-operation overhead. Therefore, it is essential that consecutive logging operation requests be *properly merged* so as to effectively amortize these per-operation overheads and still rein in the average logging latency. Third, to make the best of the raw data transfer capability of modern disks, it is absolutely crucial to transform high-level logging operation requests to low-level disk access requests in such a way that the logging disks see an *un-interrupted* stream of disk write requests with consecutive target addresses. Without such careful planning and scheduling, the logging disks may end up sitting idle most of the time.

Rising to these challenges, we devise a novel disk logging system architecture called *Beluga*, which features a *floating* logging operation API that allows an application to perform a logging operation without specifying the target address of the operation's payload, and a highly streamlined disk write pipeline that aggregates logging operation requests optimally and moves aggregated operations through the pipeline in such a way that makes full use of the disk's raw data transfer capability. Empirical measurements on a fully operational 3-disk *Beluga* prototype show that it can achieve 1.2 million 256-

byte logging operations per second, with each logging operation's latency kept below 1 msec. Moreover, even when logging operation requests arrive sparsely, *Beluga* is still able to achieve sub-msec logging operation latency. To the best of our knowledge, this is the best disk logging performance ever reported in the literature.

Beluga is designed as a building block for constructing high-level logging and recovery subsystems, and provides the following service abstraction to its applications such as BOSC-based storage system, file system or DBMS: a cyclic persistent log device which is large enough (tens of gigabytes) that FIFO-based garbage collection works adequately. That is, by the time *Beluga* reaches the end of an application's log device, the log records in the beginning of the log device are no longer needed and therefore *Beluga* could wrap around and continue logging from the beginning. In terms of functionalities, at run time, *Beluga* synchronously writes the payload of each logging operation to disk and at recovery time, *Beluga* retrieves the active portion of a recovering application's log device and returns them to the application. However, *Beluga* cannot interpret the payloads of retrieved log records because the size and structure of each application-specific log record is completely opaque to *Beluga*. Instead, it is the application's recovery subsystem that performs such interpretation on the log records returned by *Beluga*. Similarly, a *Beluga* application needs to decide what information to log, e.g. metadata updates or checkpoint summary and then utilizes *Beluga*'s service to log them to disk. In summary, *Beluga* writes log records to disk and reads them back at recovery time; how the log records are composed and how they should be interpreted are up to the applications using *Beluga*.

Although solid-state disk (SSD) is a promising technology for disk-intensive workloads, it is not necessarily a better fit than hard disks (HDD) for logging operations for the following reasons. First, because mainstream SSDs start to use multi-level cells, the per-cell write count limit is reduced to 10000, which may not fare well with the write-intensive nature of logging operations. Second, HDDs command a significant per-byte cost advantage over SSDs and make it more feasible to trade space for performance by giving abundant space to each log device so as to reduce the garbage collection overhead to the minimum. Through this research, we demonstrate that HDD-based disk logging is still among the most cost-competitive choice for logging applications.

6.1 Vanilla Disk Logging

Our first attempt was to write a user-level Linux application that synchronously appends the payload of every incoming logging operation request to a log file.

File/Raw	Threads	Latency	Throughput
File	1	14.149	25.3
File	8	14.125	25.3
Raw	1	8.308	119.8
Raw	8	8.312	119.8

Table 6.1: *Latency and throughput of file-based and device-based (Raw) disk logging. The logging operation request size is 512 bytes. Throughput is measured in terms of number of logging operations per second and latency in milliseconds.*

As shown in Table 6.1, where the logging payload size is 512 bytes, the average logging latency of this disk logger is 14.1 msec and its logging throughput on a single disk is lower than 30 logging operations per second even with 8 threads issuing logging operation requests concurrently. A closer investigation reveals that file-based logging entails several drawbacks. First, file system caching introduces latency penalty due to extra data copying. This problem could be lessened by specifying the `O_DIRECT` option while opening the log file, which bypasses file system caching altogether. Second, each logging operation is embodied by a file write system call, which could trigger multiple disk I/Os because of accesses to file system metadata. Our measurement suggests that on average 5.2 disk I/Os are triggered by each file-based logging operation. Third, because the basic data read/write unit of the file systems is 4KB and caching is disabled for reasons mentioned above, a file system write operation may require a read of the disk block containing the write operation's target address range before the write if the logging operation payload is smaller than 4KB.

Our next attempt was to implement the logger as a Linux application that synchronously appends the payload of every incoming logging operation request to a raw device. This device-based logging design removes the first two problems of file-based logging by construction. The average number of disk I/Os per device-based logging operation is exactly 1. It also mitigates the third problem by using 512 bytes as its minimum data read/write unit. But, if the logging operation request size is smaller than 512 bytes, either an additional read is still needed or one disk sector is used in each logging operation. As shown in Table 6.1, device-based disk logging increases the logging throughput on a single disk to 120 logging operations per second when 8 threads are used. However, the average logging latency is still quite high, 8.3 msec. This high latency mainly comes from the fact that consecutive logging operations are issued synchronously. More concretely, the $N + 1$-th logging operation is issued only after the N-th logging operation is completed. This

means that by the time the disk I/O for the $N+1$-th logging operation reaches the disk, it misses its target sector and needs to wait for a full rotation, which is roughly 8.3 msec for a 7200 RPM disk drive.

Both device-based and file-based logging are based on top of Linux's block I/O layer, which abstracts the physical block I/O devices and presents a unified interface to high-level software. The block I/O layer maintains a request queue for each device to hold incoming block I/O requests destined to that device and whenever possible merges every incoming request with some existing requests already in the queue if their target address ranges are adjacent to each other. In addition, the block I/O layer also re-orders requests in the queue to either improve the device's throughput or deliver differentiated quality of service (QoS) to different processes.

6.2 Toy-Train Disk Logging

Beluga is a highly efficient disk logging system designed to address all the deficiencies described earlier and has successfully been implemented in the Linux kernel 2.6.39.1.

6.2.1 Conceptual Model

Our objective is to translate the raw data transfer bandwidth of modern disks into high throughput and low latency for logging operations. Towards this goal, we develop a *toy-train disk logging* technique, which constantly submits new disk write requests with consecutive target disk addresses to the logging disk so as to keep the disk fully occupied, *even in the absence of application-level logging operation requests*. This disk logging model makes it possible for the disk I/O software to have a tight grip of the disk head position without requiring intimate knowledge of the disk internals as in such previous efforts as Trail [107]. The proposed disk write pipeline is analogous to a toy train moving constantly around a closed circuit with two stations, with cargo uploaded in one station and offloaded in another. Even when no cargo is aboard, the train is still running around the circuit at full speed. In addition, this train actually never stops, because it can upload and offload cargo on the fly without slowing down. Since *Beluga* builds such a tightly controlled pipeline, it needs a dedicated logging disk without sharing it with other data workloads.

The throughput and latency performance of the proposed toy-train disk logging mechanism is excellent when the input load is dense (input logging request queue is full all the time) and sparse (input logging request queue is empty most of the time). However, when the input load is sparse, *Beluga*

raises two concerns. First, *Beluga's* constant disk writing model increases the wear of disk platters. Second, *Beluga's* constant disk writing model increases the power consumption of logging disks. The first concern turns out to be a non-issue because of the huge capacity of modern disks. For example, given a 2-TB disk, if *Beluga* is able to write 100MB/sec, it means a given disk sector is overwritten once every 20000 (2TB/(100MB/sec)) seconds, or fewer than 5 times per day, or fewer than 2000 times per year, or fewer than 10000 times per 5 years, a disk's typical expected life time. Therefore, the level of wear induced by *Beluga* is well within the wear limit of modern hard disks.

The power consumption concern, however, is justified, because a disk's power consumption differs significantly when the disk head is idle and when it performs a read/write operation [192]. To reduce the additional power consumption when the input load is sparse, we develop a low-power version of *Beluga*, whose details are explained in Section 6.2.5.

6.2.2 Application Programming Interface

Beluga is designed to be a server's disk logging subsystem that supports logging operations from applications running on the same server. Its application programming interface (API) consists of the following three functions:

- `log_open(log_name)` takes a character string as input and returns a descriptor to a new log device, allows an application to give its log a symbolic name and associates the log's name with a log device's descriptor that later is used in logging operations.

- `log_write(device_descriptor, buffer_ptr, length)` appends the byte sequence defined by `buffer_ptr` and `length` to the end of the log device denoted by `device_descriptor`.

- `log_read(device_descriptor, buffer_ptr, length)` allows an application to read backwards from the end of the log file denoted by `device_descriptor` argument a byte sequence of size `length` to the buffer area pointed to by `buffer_ptr`.

To minimize data copying overhead for fine-grained (smaller than 512 bytes) disk logging, *Beluga* bypasses the file system and raw device access interfaces available in Linux and uses the `ioctl` interface to implement `log_read()` and `log_write()`. More concretely, the payload of a logging or log write operation is passed directly to a *Beluga* kernel module above Linux's block I/O subsystem after entering the kernel. Upon receiving a logging operation's payload, *Beluga* prepends a *management header* to it, inserts the result into an

accumulation queue and later on at an appropriate moment submits the accumulated result as a single disk write request to Linux's block I/O subsystem. The per-log-record management header is application-independent and contains the log record size, a valid bit, the associated log device descriptor and a per-device sequence ID. The valid bit holds 0 if a record is dummy or 1 if its issued from an user application. The actual contents of the log records are opaque to *Beluga*. With this arrangement, *Beluga* could combine the payloads of logging operations to different logging devices, each potentially using a *different log record size*, into a single physical disk write request and submits it to Linux's block I/O layer.

Beluga multiplexes multiple log devices, each associated with a different application, onto a physical disk, possibly in an interleaved fashion. When a server equipped with a *Beluga* disk logging subsystem fails and restarts, for each log device *Beluga* first identifies the youngest log record by performing a binary search of the underlying logging disk(s) based on the per-log-record management headers and then waits for the applications' recovery components to read back their respective log devices for further recovery processing. *Beluga* itself does not interpret log records and therefore is not involved in application-level recovery processing other than retrieving the requested log records.

Beluga assumes that the log device of each logging application is provisioned with sufficient storage space that the application could simply treat its log device as a cyclic FIFO buffer without applying any garbage collection mechanism. For example, a 500GB log device could hold the payloads of up to 1 billion 512-byte logging operations without overwriting any old log records, which is large enough to allow *Beluga* to simply wrap around a log device when it reaches the device's end. To accommodate applications that need longer retention period for their log records, one needs to provision atleast two disk arrays, so that while *Beluga* is logging on one disk array, one can archive logging records in the other disk array to another medium.

6.2.3 Streamlined Disk Write Pipeline

A major optimization goal of the *Beluga* project is to convert as much as possible a disk's raw data transfer rate into a proportionally high I/O rate, e.g., turning a byte rate of 100Mbytes/sec into an I/O rate of 100000 1KB-writes per second. The key to enable such efficient conversion is to feed the disk with write requests with consecutive start addresses in such a way that the on-disk controller constantly puts data onto the disk platters *nonstop*. Only disk drives that support command queuing could service one request after another without gap between them. Most modern SATA drives come with an efficient command queuing implementation called Native Command

Queuing (NCQ) [193]. NCQ provides three optimization mechanisms. *First*, it queues disk access commands in the disk drive and makes it possible for the on-disk controller to immediately service the next command in the queue when the previous command is completed. *Second*, NCQ batches and/or schedules queued commands to reduce the number of commands that need to be serviced and the disk access overhead. *Third*, NCQ also supports interrupt coalescing, which aggregates multiple completion interrupts and signals the host once for them to reduce the total interrupt processing overhead.

The keys to maximize the logging operation rate are (a) properly batching incoming logging operation requests to balance between latency and data transport efficiency and (b) constantly moving data to the disk platter. To embody these two ideas, we devise a four-stage pipeline to process fixed-sized disk write requests, as shown in Figure 6.1. In the first stage (*Accumulate*), high-level logging operations are inserted in the aggregate queue and aggregated into low-level disk write requests. In the second stage (*Submit*), aggregated disk write requests are copied from the host memory to the on-disk queue managed by NCQ. In the third stage (*Transfer*), the payload of a queued disk write request is transferred to its associated location on the disk platter. In the fourth stage (*Complete*), the disk delivers a completion interrupt to the host for every completed disk write request, which in turn triggers additional processing on the host to complete each high-level logging operation associated with the completed disk write request. In this pipeline design, the on-disk controller takes care of the second half of the *Submit* stage, the *Transfer* stage and the first half of the *Complete* stage and the rest is fully controlled by the host software. Because the on-disk controller is opaque to the host software, the cycle time of this pipeline is mainly determined by the *Transfer* stage. Of course, the time taken by the *Transfer* stage depends on the size of the disk write request's payload. So a major design issue here is to determine the optimal disk write request size so that the four stages in this pipeline are balanced.

When the payload of the N-th disk write request is fully transferred to the disk platter, the on-disk controller starts the transfer of the $N + 1$-th request's payload to the disk platter and sends a completion interrupt to the host, which arranges a DMA to move the payload of the $N + 2$-th request's payload into the disk. If the $N + 2$-th request's payload does not reach the disk in time, i.e., before the transfer of the $N + 1$-request's payload is done, the on-disk controller won't be able to transfer the $N + 2$-th request's payload immediately after completing the transfer of the $N + 1$-th request's payload, thus wasting a full rotation delay. Careful selection of optimal disk request size avoids such a scenario and ensures smooth transitions in the pipeline.

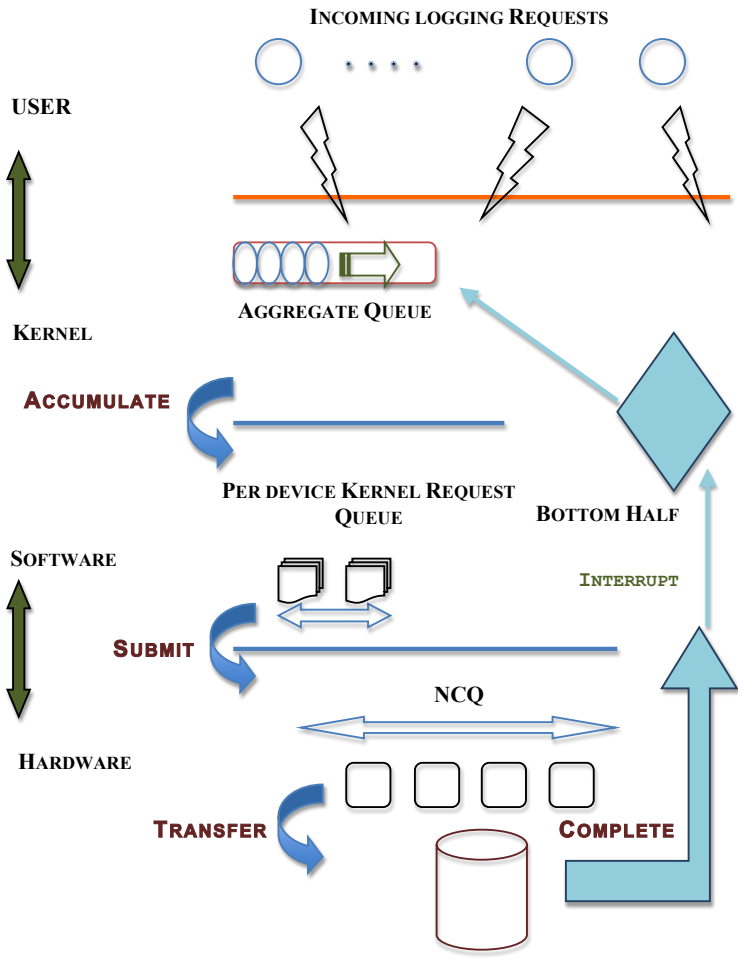

Figure 6.1: *The proposed four-stage disk write pipeline includes Accumulate, Submit, Transfer and Complete and the queues involved are aggregate queue, per-device kernel request queue in the block I/O layer and on-disk NCQ queue.*

To avoid such waste, one must minimize the critical path, which includes the interrupt generation on the disk, the interrupt processing on the host and the payload DMA. To minimize the interrupt generation time, NCQ's interrupt coalescing must be disabled. To minimize the impact of the interrupt processing time, the host software must be able to schedule the payload DMA as soon as possible after receiving the hardware interrupt. Although the raw data rate of modern PCIe bus (Gen2 or Gen3) is higher than that of the disk transfer bandwidth, the granularity of each disk write request must be sufficiently high to amortize the non-trivial fixed overhead that each PCIe bus transaction incurs. We discuss the optimal granularity or *batch size* for disk write requests in the Performance Evaluation section.

Because NCQ itself could also batch and schedule disk write requests in the on-disk queue, it could potentially increase the time taken by the *Transfer* stage by batching adjacent requests, or destroy the sequentiality of requests serviced consecutively because of its rotation delay-aware scheduling. Suppose there are five 64-KB disk write requests in the on-disk queue. It is possible that NCQ's scheduling logic may choose to service the fifth request after servicing the first request because the fifth request is closer to the first request than the second request. However, doing so seriously disrupts the pipeline. Therefore, among the three optimization mechanisms that NCQ provides, *Beluga* only wants its command queuing mechanism, and has to do away with the other two mechanisms.

6.2.4 Dense-Mode Logging

In the process of designing *Beluga*, we distinguish between *dense-mode logging*, in which high-level logging operations arrive at the logging subsystem at a rate equal to its maximum throughput and *sparse-mode logging*, in which high-level logging operations arrive at the logging subsystem slower than its maximum throughput. The design goal for dense-mode logging is both high logging throughput and low logging latency, whereas the design goal for sparse-mode logging is low logging latency only.

When *Beluga* receives high-level logging operations, it queues them in its buffer area and batches them into low-level aggregated disk write requests of optimal batch size. The optimal batch size for a disk write pipeline on a server depends on its PCIe bus and disk interface (SATA, SAS or SCSI), which affect the *Submit* stage time and the RPM rating of the disk, which affects the *Transfer* stage time. In fact, because the *Transfer* stage time for a disk write request of a certain size may vary depending on where its target address lies on the disk surface, the optimal batch size of a disk write pipeline changes as the pipeline traverses different parts of the disk surface. More specifically,

because outermost tracks (closer to track 0) have higher sector density than innermost tracks, the former's data transfer rate is higher than the latter's. Therefore, the optimal batch size is expected to increase as the target addresses of the disk write requests issued by *Beluga* increase. Then when a completion interrupt arrives, *Beluga* dispatches an aggregated disk write request to Linux's block I/O subsystem. To prevent the merging and scheduling functionalities of Linux's block I/O subsystem from getting in the way, *Beluga* turns on the *no-merge* option and chooses the *Noop* I/O scheduler. Essentially, *Beluga* does its own buffering and batching according to the streamlined disk write pipeline design and uses Linux's block I/O subsystem as a dumb pipe to push the aggregated disk write requests through to the disks.

To turn off the scheduling functionalities of NCQ, *Beluga* sets the NCQ queue length to 2, so that the on-disk controller does not have more than one choice at a time.

An implementation challenge for dense-mode logging is how to shorten the critical path described earlier, more specifically, how to quickly transfer the payload of the disk write request that is the next to be dispatched to the disk. The Linux kernel uses a separate kernel thread to dispatch the next disk write request and transfer its payload to the on-disk queue. This implementation introduces additional delay because after a completion interrupt arrives at the host, control goes through the interrupt handler, the OS's scheduler, the software interrupt logic, possibly other kernel threads and eventually the request-dispatching kernel thread. To remove these delays in this implementation, we attempted to modify the Linux kernel so that it dispatches the next disk write request from the aggregate queue *directly* in the context of the completion interrupt handler, thus more tightly tying a completion interrupt with the payload movement it triggers. While conceptually straightforward, it requires a non-trivial modification because in Linux, functions that could potentially block, such as `submit_bio`, are not supposed to be called from an *atomic* context, because such an execution context does not persist and thus is not supposed to block. Eventually we decided to stay with the original kernel thread-based implementation.

In *Beluga*, the target logical block address of an aggregated disk write request is determined only at the point when it is dispatched. *Late address binding* is necessary especially when multiple physical disks are used in a *Beluga* based disk logging system, where the relative timing for request completion among these disks could vary due to run-time conditions and thus is not fully deterministic. Accordingly, the target logical block address of each high-level logging operation is also determined only when its associated aggregated disk write request is dispatched.

After a disk write request is completed, *Beluga* demultiplexes this completion signal to the logging operations that compose the disk write request by invoking their corresponding post-completion request completion logic. The latency of a logging operation is the time interval between when the logging operation enters *Beluga*'s buffer and when the post-completion processing of the logging operation is finished.

To jump-start the proposed streamlined disk write pipeline, *Beluga* first issues *back to back* two disk write requests to the disk to fill up the *Transfer* and *Submit* stage and then holds off the third disk write request until the completion interrupt of the first disk write request arrives. After that, a new aggregated disk write request is fed to the pipeline only after an existing disk write request exits the pipeline.

It is straightforward to generalize the above design to multiple logging disks, because Linux allocates a separate per-device request queue for each individual disk. However, the *Accumulate* stage is centralized and *Beluga*'s accumulation buffer is shared by the logging disks. That is, *Beluga* aggregates incoming high-level logging operations into low-level disk write requests for all logging disks and as soon as a completion interrupt from a logging disk comes, it dispatches an aggregated disk write request to that particular logging disk. To avoid contention among logging disks, the logging disks are jump-started in a staggered fashion to prevent unwanted synchronization among them.

The optimal batch size for a disk write pipeline on a server depends on its PCIe bus and disk interface (SATA, SAS or SCSI), which affect the *Submit* stage time and the RPM rating of the disk, which affects the *Transfer* stage time. In fact, because the *Transfer* stage time for a disk write request of a certain size may vary depending on where its target address lies on the disk surface, the optimal batch size of a disk write pipeline changes as the pipeline traverses different parts of the disk surface. More specifically, because outermost tracks (closer to track 0) have higher sector density than innermost tracks, the former's data transfer rate is higher than the latter's. Therefore, the optimal batch size is expected to increase as the target addresses of the disk write requests issued by *Beluga* increase.

In summary, for dense-mode logging, *Beluga* accumulates user-level logging operation requests, aggregates them into physical disk write requests, feeds them into a streamlined disk write pipeline and delivers a completion response to each user-level logging operation when its payload reaches the disk. This streamlined disk write pipeline is novel because it is designed to move fixed-sized data payload in a lock-step fashion, similar to a CPU pipeline, so as to fully exploit the disk's raw data transfer capability and effectively convert its data transfer rate (Mbytes/sec) into the commensurate I/O rate (IOs/sec).

6.2.5 Sparse-Mode Logging

An implicit assumption underlying the design of the streamlined disk write pipeline is that there is an infinite stream of disk write requests that are waiting to fill the pipeline. This assumption is valid for dense-node logging, but does not hold for sparse-mode logging. More concretely, if a logging operation request appears after a period of inactivity, this logging operation request enters the disk write pipeline *alone* and therefore cannot benefit from any disk head position information that may be gleaned from neighboring requests, as is the case in dense-mode logging. As a consequence, the average latency of such logging operation requests may be high, because it is difficult to ensure that the target address assigned to a sparse-mode logging operation request is close to the disk head position at the time when the request is submitted.

One way to reduce the latency of sparse-mode logging operations is to constantly predict the disk head position [107], and use this prediction to derive a better target logical block address for each sparse-mode logging operation. However, as modern disks become more and more complicated, this approach becomes less and less effective, because the internal control mechanisms inside disk drives, such as NCQ, on-disk caching, interrupt coalescing, etc., tend to obscure disk head movement and thus get in the way of disk head position prediction.

In contrast, *Beluga* leverages its dense-mode logging architecture to implement sparse-mode logging. More concretely, *Beluga* constantly maintains a dummy aggregated disk write request in its buffer and dispatches this request to the disk write pipeline when it runs out of application-level logging operations. However, whenever *Beluga* occasionally receives an application-level logging operation request, it aggregates this application-level logging operation request to the dummy disk write request currently being formed. When the next disk completion interrupt comes, *Beluga* dispatches this disk write request as usual. That is, *Beluga* keeps the disk write pipeline constantly busy either with real disk write requests accumulated from application-level logging operation requests (dense-mode logging), or with dummy disk write requests (sparse-mode logging), some of which may contain high-level logging operations issued by applications.

The key advantage of this design is that it is self-adaptive to the timing variations of the disk write pipeline. That is, because the disk write pipeline is driven by events such as request completion interrupts rather than by a hardware clock, the timing experienced by each disk write request may vary. However, by keeping the disk write pipeline full with dummy write requests, all the timing variations due to firmware, software or hardware are automatically accounted for and thus removed from the implementation complexity of sparse-

mode disk logging. The main drawback of this design is the additional power consumption associated with dummy disk write requests.

The original *Beluga*'s design (called *full Beluga*) continuously dispatches disk write request of size S whose target addresses are S apart, where S corresponds to the batch size. There are two possible approaches to reducing the power consumption of this design. The first approach is to submit the same sequence of disk write requests in the same way as in full *Beluga* but decrease the size of each submitted disk write request to one disk sector (512 bytes), when there are no pending logging operation requests. This way, the number of bytes written to disk could be reduced by one to two orders of magnitude when there are no pending logging operation requests. The second approach is to issue only $\frac{1}{N}$ of the disk write requests in full *Beluga*, employ the issue times and completion interrupt times of these disk write requests to estimate the issue times of the skipped $\frac{N-1}{N}$ disk write requests had they been dispatched in full *Beluga*, and use the estimated issue time for logging operation requests that arrive sparsely. This way, the number of bytes written to disk is reduced by a factor of N when there are no pending logging operation requests.

Unfortunately neither approach works as expected because some modern disks implement a *request merging* optimization: When a disk write request R arrives at a disk's NCQ queue and cannot be merged with any existing disk write request, the on-disk controller tries to merge R with some future requests by deferring servicing R for approximately 1 msec after it arrives, even if the disk head passes R's target address within this 1-msec interval. This optimization gets in the way of the above two approaches because none of the disk write requests issued in these two designs are mergeable with any existing disk write request. In addition, the on-disk controller also implements a request scheduling mechanism, which makes it difficult to predict the actual service order of disk write requests even when their target addresses are sorted.

To get around the request merging mechanism, the target address of each submitted disk write request is set at 1 msec away from the disk head position at the time when it arrives on the disk. This prevents each submitted disk write request from experiencing a full rotation delay. To get around the on-disk request scheduler, we limit the effective number of active requests in the NCQ queue to 4 and ensure that they are sufficiently far apart. Taking into account these design constraints, we come up with a low-power version of *Beluga* as follows. A sequence of *sentinel* disk write requests are dispatched to the logging disk regardless of whether applications issue any logging operation requests. When the N-th sentinel request is completed, low-power *Beluga* issues the $N+2$-th sentinel request. The distance between the target addresses of consecutive

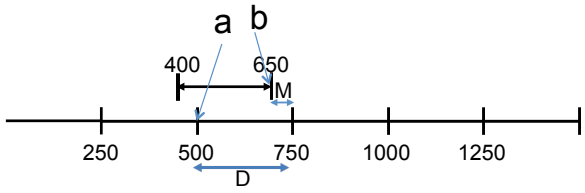

Figure 6.2: *An example sentinel disk write request schedule for low-power Beluga when the inter-sentinel-request distance is 250 sectors and the margin is 100 sectors. Logging operation requests issued by applications during an interval are aggregated into a disk write request that is merged with the corresponding sentinel request.*

sentinel requests is D sectors, where the time it takes for the disk head to pass D sectors is at least 1 msec. Suppose the target address of a sentinel request is Sector T, then all application-issued logging operation requests that arrive between the time when the disk head passes Sector $T - M - D$ and the time when the disk head passes Sector $T - M$ are aggregated into one disk write request that is to be merged with this sentinel request. The interval marked by these two time points is the *feasible interval* associated with this sentinel request. M is an empirical safety margin in the following sense: If a new request is to be merged with an existing sentinel request without disrupting the service order, the new request must arrive at the disk at least M sectors before the disk head passes the existing request's target address. The size of each sentinel request is 4KB because this is the minimum size for a request to be mergeable. This low-power *Beluga* design not only dispatches fewer disk write requests than full *Beluga*, but also keeps each request smaller than those in full *Beluga*.

Figure 6.2 shows an example schedule of sentinel requests in low-power *Beluga*, where D is 250 and M is 100. The target addresses of the sentinel requests are Sector 250, 500, 750, 1000, 1250, 1500, etc. The sentinel request with the target address Sector 750 is dispatched when the request with the target address Sector 250 is completed, or about 500 sectors before it is serviced. It takes more than 1 msec for the disk head to fly over 500 sectors. In addition, at most two sentinel requests are in the on-disk queue at a time. For this sentinel request (spanning Sector 750 to 757), all application-issued logging operation requests that arrive between the time when the disk head passes Sector 400 and the time when the disk head passes Sector 650 are aggregated into one disk write request whose target address is Sector 758 and submitted to the disk when the disk head passes Sector 650. If there are no application-issued logging operation requests, the number of bytes written in

low- power *Beluga* is 8 sectors every 250 sectors, or roughly $\frac{1}{30}$ of that of full *Beluga*.

6.3 Performance Evaluation

We have successfully built a Linux-based *Beluga* prototype and carried out a detailed performance evaluation of this prototype using a Dell Machine with a 1.8GHz Dual-Core processor, 2 GB Memory and three 1 TB WD Caviar Black 7200RPM SATA hard disks and a Gen2 PCIe bus.

6.3.1 Methodology

A sequence of logging operation requests were fed to the *Beluga* prototype and the average logging latency and overall logging throughput were measured. The latency of each logging operation request is the time when it enters the logging subsystem and when the logging subsystem returns a completion response for it. Throughput is defined as the total number of logging operations completed, i.e., its payload is written to disk, per second. We used an open-source tool on Linux called *blktrace* to take these measurements. *Blktrace* is a block-layer I/O tracing tool that provides detailed information on where the time is spent on the data path from the entry point to the block I/O layer until data is written to the disk and a completion interrupt is delivered.

To measure the maximum throughput of *Beluga*, we develop a kernel-level logging operation traffic generator that minimizes the overhead of creating application-level logging operation requests. The *kernel level log generator (KLG)* is installed as a loadable kernel module. Its preferred over a real-world user-level application because the logging workload from KLG is the most demanding and could best stress *Beluga*.

We modify the kernel scheduler so that immediately after the main thread of *Beluga* sets up a DMA to move logging operation payloads to the disk, it calls KLG, which generates logging operation requests according to the elapsed time since it was invoked and the input traffic load and put them in the input queue. The *Beluga* thread then moves as many requests in the input queue to its aggregate queue as possible, whenever it has a chance to visit the input queue. Moreover, to minimize the interference between the KLG and the logging subsystem, we bind the KLG to a specific set of CPU cores, which are disjoint from those on which *Beluga* runs. The *Beluga* prototype has been stress tested with continuous input logging operations to cover all disk blocks on the log disk multiple times. However, in each experiment run, we issued 60 seconds worth of logging operations. Its also stress tested to ensure other sys-

tem activities don't disturb *Beluga*'s tightly controlled pipeline. We ran several memory intensive and disk I/O intensive programs in parallel and observed that *Beluga*'s performance remained unaffected to a large extent. Different runs correspond to different combinations of batch size, logging operation request size and inter-operation interval. Although *Beluga* aggregates logging operations into disk write requests, the average logging latency is the end-to-end delay experienced by each logging operation and the logging throughput corresponds to the number of logging operations completed per second.

6.3.2 Dense-Mode Logging

Overall Performance

The most important parameter in the proposed disk write pipeline is the *batch size*, or the granularity of the fixed-sized disk write requests that are moved through the pipeline. When the batch size is too small, the time to submit a disk write request from host memory to on-disk queue is longer than the time required to transfer it from on-disk queue to the disk platter, because of the non-trivial per-transaction overhead and as a result, by the time the disk write request reaches the disk, the immediately previous disk write request is already done and it misses its target sector and is thus delayed by a full rotation cycle. When the batch size is too high, each logging operation request experiences a higher queuing delay in the accumulation queue and the transfer time is also higher; consequently the average logging latency is higher.

Table 6.2 shows the average logging latency and logging throughput for an input sequence of 10 million 256-byte logging operations under different batch size. When the batch size is 16KB, the *Submit* stage time is longer than the *Transfer* stage time and almost every disk write request experiences a full rotation cycle time. When the transfer delay of a disk write request is increased to a full rotation delay, the aggregation delay of the logging operations in the following disk write request is also increased on average by half of the transfer delay. As a result, the average logging latency is quite high, 12.7 msec, and the logging throughput is accordingly low, 15044 operations per second, the inverse of which corresponds to the average data transfer delay.

As the batch size is increased from 16KB to 24KB and 28 KB, the probability of an aggregated disk write request experiences a full rotation delay decreases but is still non-zero. As a consequence, the average logging latency also decreases and the logging throughput increases. When the batch size reaches 32KB and beyond, the *Submit* stage time is always smaller than the *Transfer* stage time and *none* of the disk write requests experience a full rotation delay. The effective pipeline cycle time is the maximum of these two

stage times. After the batch size grows larger than 32KB, the average logging latency is worsened, because the initial aggregation delay is higher and the pipeline cycle time is increased, which leads to longer end-to-end pipeline latency. However, the logging throughput improves with the batch size, because the fixed overhead associated with the *Submit*, *Transfer* and *Complete* stage is more efficiently amortized. Because the design goal of *Beluga* is both low logging latency and high logging throughput, the ideal batch size is the smallest batch size that enables the *Submit* stage time is smaller than the *Transfer* stage time and the default batch size used in the current *Beluga* prototype is 32KB.

Batch Size	**Latency(μsec)**	**Throughput(OPs/sec)**
16 KB	12753	15044
24 KB	3087	108471
28 KB	2832	118157
32 KB	938	404228
40 KB	1108	428005
48 KB	1327	429665
56 KB	1536	433331
64 KB	1755	433990

Table 6.2: *Average latency and throughput of 256-byte logging operations on the Beluga prototype when the batch size is varied from 16KB to 64KB*

To examine where each logging operation spends its time in the *Beluga* prototype, we used *blktrace*, which is an analysis and instrumentation tool for Linux's block I/O layer. *Blktrace* breaks the data path from Linux's block I/O layer down into stages and give timing measurements(in μsec) for each of them. The following are definitions of a set of terms used in our analysis:

- Aggregation Delay: the amount of time a logging operation stays in the aggregate queue.
- Q2D: Time required for an aggregated request to be inserted into and to stay in the per-device queue.
- D2C: Time between when a disk write request is issued to a disk and when it is completed, as indicated by a completion interrupt delivered to the block I/O layer.
- Q2Q: Time between two consecutive aggregate disk write requests that are inserted into the block I/O layer's request queue.

Table 6.3 shows the detailed breakdown of the time a logging operation spends in the disk write pipeline when the batch size is varied. Because *Beluga* aggregates logging operation requests into disk write requests, the latency experienced by a logging operation request includes the time it spends in the aggregate queue (Aggregate Delay) and the latency experienced by the disk write request to which it belongs (Q2D + D2C). D2C includes the *Submit* stage time and the *Transfer* stage time. The *Submit* stage time includes the completion interrupt processing time, which is about 100 μsec and the data transfer time on the PCIe bus. Because *Beluga* disables the merging, sorting and the scheduling mechanisms in Linux's block I/O layer, Q2D is very small, less than 2 μsec. Q2Q corresponds to the cycle time of the disk write pipeline and its inverse corresponds to the pipeline's throughput.

When the batch size is 16KB, D2C is 8500 μsec, which suggests that every aggregated disk write request misses its target sector when it arrives at the disk and experiences a full rotation delay, about 8.3msec for a 7200RPM disk drive, because the *Submit* stage time is higher than the disk data transfer time of 16KB, about 160 μsec. Moreover, the Aggregation Delay is 4251 μsec, which is higher than expected and is a collateral damage of the longer disk data transfer time. When the batch size is 28KB, D2C is decreased to 1786 μsec, which suggests that still a certain percentage of disk write requests miss their target sector and experience a full rotation delay and the Aggregation Delay is decreased accordingly to 944 μsec.

When the batch size is 32KB, D2C is 523 μsec, the Aggregation Delay is 313 μsec and none of the disk write requests experience a full rotation delay. Because Q2Q or the pipeline cycle time is 316 μsec, the throughput of this configuration is 3158 32KB disk write requests per second or 404228 256-byte logging operations per second, as shown in Table 6.2 . Because the *Transfer* stage time is larger than the *Submit* stage time, the *Transfer* stage time is the pipeline cycle time. Therefore, within DC2, 316 μsec is due to data transfer (Q2Q), 100 μsec is due to interrupt processing and the remaining time (107μsec) is due to data transfer on the PCIe bus. Because there is noticeable difference between the *Transfer* stage time and the *Submit* stage time, the optimal batch size, which corresponds to the case when the *Transfer* stage time is the same as the *Submit* stage time, lies somewhere between 28KB and 32KB. Nonetheless with the batch size of 32 KB, *Beluga* delivers an average logging latency of under 1 millisecond and a logging throughput of 404K 256-byte logging operations, which exceeds 100 Mbytes/sec and is pretty close to the raw disk data transfer capability.

When the batch size is 40KB, the pipeline cycle time is increased to 373 μsec, but the throughput is also increased to 428K logging operations per

second, because a 40KB aggregated disk write request contains more logging operations than that in a 32KB aggregated disk write request. Unfortunately, the average logging latency is also increased to 1108 μsec, partly because the Aggregation Delay is increased to 369 μsec.

Batch Size	**Aggregation Delay** (μsec	**Q2D** (μsec)	**D2C** (μsec)	**Overall Latency** (μsec
16 KB	4251	1.871	8500	12753
24 KB	1029	1.638	2056	3087
28 KB	943	1.625	1887	2832
32 KB	313	1.719	623	938
40 KB	369	1.829	737	1108
48 KB	442	1.688	883	1327
56 KB	512	1.904	1022	1536
64 KB	585	1.768	1168	1755

Table 6.3: *Detailed breakdown of the time each logging operation spends in the disk write pipeline as the batch size is varied*

Logging Operation Size	**Latency** (μsec)	**Throughput (OPs/sec)**
512 B	959	193275
256 B	938	404228
128 B	986	849461
64 B	1011	1639408

Table 6.4: *The average logging latency and logging throughput for a sequence of logging operations when the logging operation request size is varied from 64 bytes to 512 bytes*

Table 6.4 shows the impact of the log operation size on the average logging latency and throughput of *Beluga*. Smaller logging operation request size only increases the overhead of aggregating logging operation requests into disk write requests, but has no effect on the disk write pipeline. This is why the average logging latency remains largely the same when the logging operation request size is varied from 64 bytes to 512 bytes. The logging throughput, on the other hand, is inversely proportional to the logging operation request size, because the disk write pipeline's throughput also stays the same.

Adaptive Batch Size Selection

Starting Offset (GB)	Optimal Batch size (KBytes)	Latency (μsec)	Throughput (OPs/sec)
900	24	1183	241158
750	24	981	290390
500	28	932	356791
0	32	938	404228

Table 6.5: *The impact of the starting disk offset used in the logging experiment on the optimal batch size for the disk write pipeline when the logging operation request size is 256 bytes*

For results reported in previous subsections, we started each experiment run at the 0th sector of the disk. Table 6.5 shows the latency and throughput of 256-byte logging operations on the *Beluga* prototype when their log records are written to different parts of the disk using different batch size. As the starting disk offset of an experiment run increases, the raw data transfer rate during the experiment run lowers, the *Transfer* stage time becomes longer and thus is more likely to be larger than the *Submit* stage time and the optimal batch size, i.e., the minimum batch size whose corresponding average logging latency is smaller than 1 msec, thus also decreases. For example, the optimal batch sizes when the starting disk offset is 0, 500GB and 750GB are 32KB, 28KB and 24KB, respectively. These batch size choices enable *Beluga* to keep the average logging latency under 1 msec. However, when the starting disk offset is 900GB, the average logging latency jumps above 1 msec regardless of the batch size because the lower disk data transfer rate at the center of the disk sets a lower bound on the latency.

Because the optimal batch size for different parts of a modern disk is different, *Beluga* includes an adaptive batch size selection mechanism that chooses the optimal batch size according to the current disk head position. This mechanism requires the log disks to be pre-calibrated so as to extract the optimal batch size for each disk region. Figure 6.3 shows that *Beluga*'s adaptive batch size selection mechanism is able to keep the average logging latency below 1 msec throughout the entire disk, whereas using a fixed batch size (in this case 32KB) could lead to an increase in the average logging latency by more than 70%, when the disk heads reach the center of the disk platters.

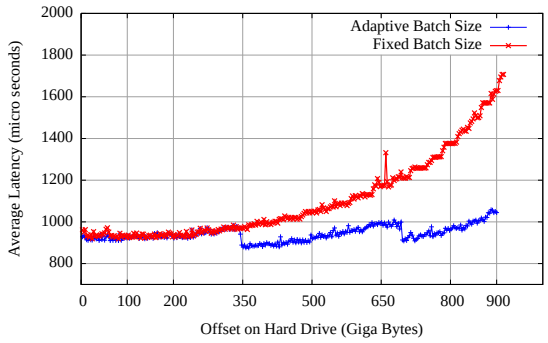

Figure 6.3: *The average logging latency of the Beluga prototype when data is written at different offsets on disk and the logging operation request size is 256 bytes*

Logging Disks	**Latency (μsec)**	**Throughput (OPs/sec))**
1	938	404228
2	934	810250
3	950	1192554

Table 6.6: *The average logging latency and throughput of the Beluga prototype when the number of disks increases from 1 to 3 and the logging operation request size is 256 bytes*

Sensitivity Study

The design of *Beluga* is linearly scalable with respect to the number of disks it uses, because each disk is equipped with a per-device request queue and an on-disk queue. However, all the disks in the *Beluga* system share a global aggregate queue, from which *Beluga* issues aggregated disk write requests to individual disks upon receiving completion interrupts. Until the *Beluga* system hits the write request issue rate limit, its logging throughput should scale linearly with the number of disks in it, as shown in Table 6.6, which also shows that the average logging latency is largely unaffected when the number of disks is increased from 1 to 3. With just three 7200 RPM SATA disks, the *Beluga* prototype is able to achieve a total logging throughput of 1.2 million 256-byte logging operations per second and keep the average logging latency under 1 msec. We believe this is the best logging performance ever reported on commodity-grade disks. The immediate bottleneck to scale up the *Beluga* prototype with even more disks is the interrupt processing overhead. Using a

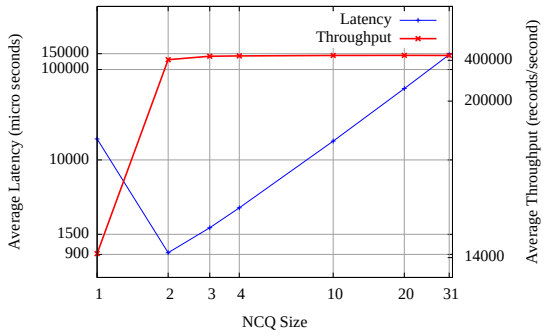

Figure 6.4: *The average logging latency and throughput of the Beluga prototype when NCQ queue length is varied and the logging operation request size is 256 bytes*

polling architecture rather than an interrupt-driven architecture is a possible solution to remove this bottleneck.

NCQ plays a key role in the design of the proposed streamlined disk write pipeline. However, the NCQ queue length is set to 2 by default, because we want to prevent NCQ's merging and scheduling functionalities from disrupting the carefully arranged timing for disk write payload movement. Figure 6.4 shows the average logging latency and throughput for 256-byte logging operations on the *Beluga* prototype when the NCQ queue size is varied. When the NCQ queue length is 1, the logging latency is very high, approximately 17msec, because there is only one disk write request in the disk at a time and it is not possible for the on-disk controller to service the next write request immediately after serving the current write request. That is, the $N + 1$-th request can be issued from the aggregate queue to the disk only after the completion interrupt for the N-th request is raised and thus has to experience at least one full rotation delay.

When the NCQ queue length is 2, the on-disk controller could service consecutive disk write requests back to back and the disk's raw data transfer capability is fully exploited. When the NCQ queue length is greater than 2, the on-disk controller still could service consecutive disk write requests back to back, but each disk write request's latency increases because it needs to wait longer in the NCQ queue. That is, when the NCQ queue length is 2, a disk write request is expected to be serviced soon after it arrives at the NCQ queue; however, when the NCQ queue length is 3, a disk write request is expected to wait for one full data transfer time before it is serviced. Moreover, the larger the NCQ queue length, the longer the average logging latency. As for the logging throughput, it remains the same as the NCQ queue length grows

to 2 and beyond, because disk write requests are serviced one after another non-stop.

6.3.3 Sparse-Mode Logging

To test the effectiveness of using full *Beluga* to support sparse-mode logging, we varied the inter-logging-operation interval and measured the average logging latency, which remains virtually constant at 938 μsec, as the inter-logging-operation interval increases from 10 μsec to 0.1 second. This shows that full *Beluga* is indeed capable of servicing sparse logging operation requests with low latency.

Distance (sectors)	**Average (μsec)**	**Re-ordering (%)**
220	8395.4	18.46
230	6835.3	16.16
240	4776.9	10.36
250	2737.3	4.62
260	2033.8	2.39
270	1324.3	0.15
280	1320.3	0.004
290	1362.7	0.001
300	1411.8	0.002

Table 6.7: *The impact of inter-sentinel-request distance on the latency of sparse logging operation requests that are dispatched immediately after one sentinel request is completed and are merged with the next sentinel request*

Low-power *Beluga* is characterized by two configuration parameters, the target address distance (D) between consecutive sentinel requests and the safety margin (M) for merging with an existing sentinel request. We conducted a series of experiments to determine the proper value of D. In each run, we dispatched a series of sentinel requests whose target addresses are spaced by D sectors and a sparse logging operation request immediately after every sentinel request is completed, which is to be merged with the next sentinel request. Table 6.7 shows the impact of D on the average latency of these sparse logging operation requests. When D is smaller than 270, the average latency is above 2 msec and the root cause is the sentinel requests are serviced out of order when the inter-sentinel-request distance is too small. For example, when D is 220, 18.46% of the sentinel requests are serviced out of order and, together

with the sparse logging operation requests that merge with them, experience a full rotation delay. The request re-ordering percentage comes down to with 1% only when D is increased to 270.

Because smaller D values lead to substantial request re-ordering, we explored the matching M value only for $D = 260$, $D = 270$, $D = 280$, $D = 290$ and $D = 300$. For each candidate D value, we tried 10 possible M values and picked the M value that results in the minimum average latency. Given a M value, between the i-th and $i+1$-th sentinel request, we issued a sparse logging operation request at the time when the disk head passes the sector that is M sectors ahead of the target address of the $i+1$-th sentinel request, measured its latency and computed the average of these latency measurements. When M is too large, dispatched sparse logging operation requests have to wait in the NCQ queue longer. When M is too small, dispatched sparse logging operation requests may cause re-ordering of sentinel requests already in the NCQ queue. Table 6.8 shows the best M value for each candidate D value. The best M value seems to be lie between 125 and 140. The *Average* column represents the average latency of sparse logging operation requests when they are dispatched to disk at the end of the feasible interval of their associated sentinel requests (Point b in Figure 6.2). In contrast, in Table 6.7 the sparse logging operation requests are dispatched to disk at the completion of the sentinel request that precedes their associated sentinel requests (Point a in Figure 6.2). That's why the Average numbers in these two tables are different.

Because sparse logging operation requests could arrive at any point of a feasible interval, they would experience a variable amount of waiting time before being dispatched to disk at the end of the feasible interval they fall in. The *E2E Average* column shows the end-to-end average latency of a sparse logging operation request, which includes this waiting time. As D increases, on the one hand the probability of request re-ordering and the associated latency penalty decreases, but on the other hand the size of the feasible interval and the average waiting time also increases. So the optimal D corresponds to a balanced trade-off between these two factors. In our testbed, the best-performing configuration is when the inter-sentinel-request distance (D) is 290 and the safety margin (M) is 125 and its end-to-end average latency for sparse logging operation requests is 1315.5 μsec. Although the average latency of low-power *Beluga* is even worse than the worst-case latency of full *Beluga* ((938 μsec), the number of bytes written in low-power *Beluga* when there are no application-issued logging operation requests is only 8 sectors (4KB) every 290 sectors, or a factor of $\frac{1}{36}$ smaller than full *Beluga*.

Distance (sectors)	Margin (sectors)	Average (μsec)	E2E Average (μsec)
260	140	1548	2089.7
270	125	768.2	1320.7
280	135	770.2	1353.5
290	125	711.3	1315.5
300	135	747.9	1372.9

Table 6.8: *The average latency for sparse logging operation requests that are dispatched at the end of the feasible interval of the sentinel requests with which they are to merge, under different inter-sentinel-request intervals and their associated margins*

Logging Operation Request Size	Latency
512 bytes	1.2 msec
2K bytes	1.34 msec
4K bytes	1.35 msec

Table 6.9: *The average logging latency of 1 million logging operations against an SSD-backed device when the logging operation request size is 512 bytes, 2KB and 4KB*

6.3.4 Comparison with SSD-based Logging

To compare the performance of the *Beluga* prototype with logging using SSDs. We measured the average latency of 1 million logging operations against an SSD-based device with the on-disk cache turned off. The SSD used in this test is a 64-Gbyte SSD based on JMicron JM612F flash controller and Samsung's SLC flash memory chips. The result, shown in Table 6.9, shows that the *Beluga* prototype's average logging latency is actually slightly better than that of SSD-based logging. Of course, the device-based logging implementation on SSD is not as extensively optimized. Actually we believe the streamlined disk write pipeline described in this work is equally effective for SSDs. Nonetheless, this result demonstrates that with proper structuring and tuning, hard disk-based logging could be as performant as SSD-based logging. In fact, for MLC SSDs, which has limited write count (around 10000), a high-performance low-latency hard disk logging technique such as *Beluga* may be a useful complement to handle sequential logging workloads.

6.4 Summary

The disk access pattern of logging is arguably the most straightforward because it is sequential in nature and yet, it is surprisingly difficult to achieve both high logging throughput and low logging latency, especially for fine-grained logging operations. The main reason is that modern I/O stacks and disk drives incorporate redundant request merging and scheduling functionalities that may get in each other's way. Moreover, although careful control of disk access timing is crucial in delivering high disk I/O performance, there is typically little coordination between the I/O stack and the underlying disks. As a consequence, the latency and throughput of vanilla file-based or device-based logging implementations are far away from the optimum. Incorporating our understanding of the root cause behind the observed performance problems, we devised a novel logging system architecture called *Beluga*, which features the following innovations:

- A logging API that supports fine-grained logging (i.e. logging payload size is smaller than a disk sector) with minimum metadata manipulation and data copying,
- A streamlined disk write pipeline that moves fixed-sized disk write requests at a constant rate while minimizing the pipeline cycle time and
- A low-power sparse-mode logging scheme that achieves low logging latency without requiring disk head position prediction.

Measurements on a fully operational *Beluga* prototype that embodies all three innovations demonstrate that using three commodity disks, the *Beluga* architecture can deliver close to 1.2 million 256-byte logging operations while keeping each logging operation's end-to-end latency below 1 msec. We believe this is the best empirical disk logging performance ever reported in the open literature. With such a high performing disk logging solution, DISCO is able to successfully integrate *Beluga* with its various data structures, to ensure better disk I/O responsiveness and guaranteed data persistency.

Chapter 7

Quality of Service Guarantee for Software-Defined Distributed Storage Systems

7.1 SDDS System Architecture in the Context of Managing QoS Functionality

A cloud storage system manages the storage requirements of the tenants' applications, enabling the tenants to not worry about managing their application's storage resources. Though a tenant greatly benefits by such a flexibility, the advantages of a cloud storage system are negated if the tenant doesn't receive satisfactory performance. For example, a cloud storage system could have a large-scale array of highly advanced flash-based SSDs, that can process I/O requests at a very high rate of 10000 I/O operations per second (IOPS). If a tenant has a real-time application that requires I/O requests to be processed no later than 1 ms, then though, on an average a large majority of the application's I/O requests are processed well within the latency requirement, since the cloud storage system isn't configured to handle strict latency requirement, some of the application's I/O requests could fail to be processed within 1 ms. Hence the tenant's application fails and the tenant is simply unhappy with the performance offered by the cloud storage system. Therefore it is an usual practice to bind performance with quality of service (QoS). QoS in a storage system can be specified by various metrics like bandwidth, in terms of either mega bytes per second (MBPS) or IOPS; latency, in terms of maximum time(micro seconds) to process an I/O request; and so on. Tenants specify these various QoS metrics using service level agreements (SLA) with the cloud storage service provider, at the time of purchasing their storage services. The cloud

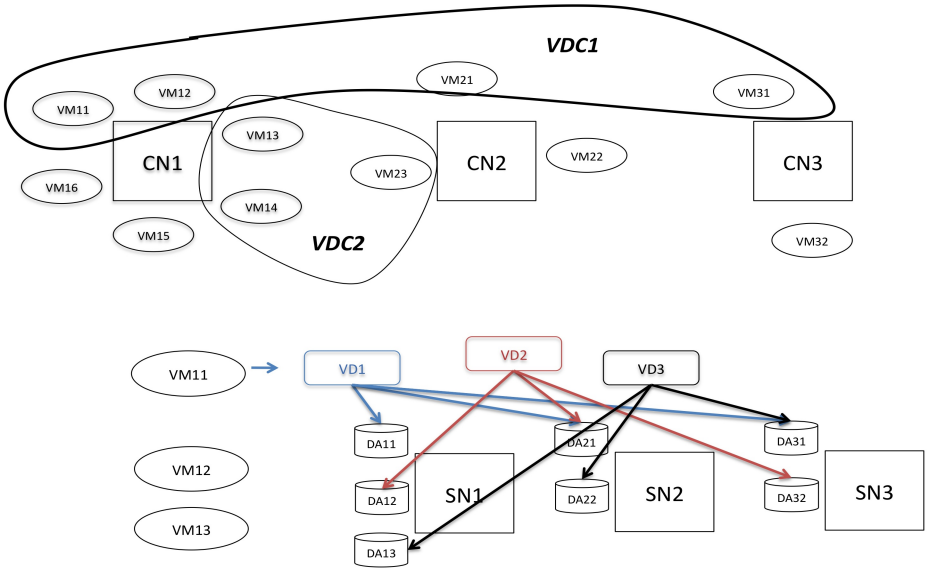

Figure 7.1: *Detailed overview of the components of a VDC*

storage service provider either accepts or rejects the SLAs, depending on the feasibility of guaranteeing the QoS requirements. Since the most important aspect of a cloud storage system is to offer efficient software services over the physical hardware resources, to ensure satisfactory storage performance, the cloud storage system is commonly referred to as software defined distributed storage (SDDS) system.

There are various challenges in designing QoS for a SDDS system, and in this work we propose a QoS model called *Cheetah*, that uses some novel techniques to enable the SDDS system to provide storage virtualization with accurate QoS guarantees. The challenges, design objectives and the novel techniques involved with *Cheetah* are better conveyed once the SDDS system's architecture is clearly defined. Section 1.2 defines the SDDS system's architecture at a much higher level and is targeted specifically to introduce the context where several pieces of this dissertation fit in together. In the following subsections, we describe the system model and service model of the SDDS system architecture in greater detail that helps understand the challenges involved in designing QoS features in the SDDS system.

7.1.1 System Model

As explained in Section 1.2, the SDDS system segregates the hardware resources into three parts: compute nodes (CN), service nodes and storage nodes (SN). A SN is essentially a commodity JBOD server, consisting of a x86 CPU and a bunch of disks that are organized into one or more sets of disk arrays (DA). A DA is just a logical collection of disks that are grouped together using software RAID techniques. All the SNs are arranged without any specific order, and the entire storage system appears as just a bunch of JBODS, which we refer to as JBOJBOD. A CN is essentially a x86 server that supports hardware assisted virtualization, and hosts several virtual machines (VM). The SDDS system provisions virtual disks (VD) in each VM, to allow tenants' applications to manage the storage requirements in a VM, such that a VD appears exactly like a physical disk to the tenants' applications. Multiple VDs can be mounted on a VM, where a VD can be a real VD like C: drive on Windows, /home directory on Unix, or a logical VD created using LVM tools on Unix. VDs are stored in entirety on any one DA. A virtual datacenter (VDC) consists of one or more VMs that are spread across various physical machines and the VDs in a VDC can be spread across several DAs, and across several SNs.

In order to support high availability, as explained in Section 1.2.3, DISCO adopts N-way replication as a default policy. With N-way replication, a disk access I/O stream from a VD is split into multiple sub streams. A write I/O request from a VD is submitted to all the N corresponding replica DAs located on N different SNs. A read I/O request is submitted to only one of the N corresponding replica DAs. Therefore, a DA receives multiple disk access I/O streams from different VDs.

7.1.2 Service Model

Cheetah is designed to support multi-dimensional storage virtualization by enforcing QoS guarantees on different aspects of the storage performance. The QoS specifications are specified as a tuple: $\langle A, B, C, D, E \rangle$, where A is availability (e.g., mean time to failure is three years, or N replicas), B is bandwidth (e.g. minimum = 250MBPS or 2500 IOPS of 1KB average request size, maximum = 500MBPS or 5000 IOPS of 1KB average request size), C is capacity (e.g. 10 TB), D is delay (e.g. worst-case delay = 20 msec) and E corresponds to elasticity with which bandwidth and/or delay metrics are guaranteed. Elasticity is expressed as the percentage of bandwidth and/or latency that has to be guaranteed.

Cheetah provides two levels of QoS granularity:

VD-level Granularity A VD is characterized with personalized QoS specifications, but if there exists a VDC that contains the given VD, then that VDC has no additional QoS specifications.

VDC-level Granularity A VDC is characterized with personalized QoS specifications, but the VDs contained in that VDC do not have any additional QoS specifications, instead all such VDs collectively share the QoS specifications of its corresponding VDC.

Though, ensuring multi-dimensional storage virtualization is the eventual goal for *Cheetah*, in this dissertation we focus only on enforcing a comprehensive VD and VDC level bandwidth guarantee. For the sake of simplicity and clarity in conveying our ideas, we also restrict high availability support to 3-way replication, but it should be noted that our proposed techniques in this dissertation can be trivially extended to support any of the other options like the generic N-way replication, parity based replication or erasure codes.

7.2 *Cheetah*'s Objectives, Challenges & Solution

7.2.1 Design Objectives

The core design objectives for *Cheetah* are:

- To strictly enforce bandwidth guarantee for each VD/VDC on a distributed storage system supporting inter-storage-node data redundancy, thereby ensuring performance isolation among the VDs/VDCs that share a DA,
- To deliver bandwidth guarantees with minimal QoS overhead, while maximizing physical resource utilization efficiency of a distributed storage system.

7.2.2 Technical Challenges

The fundamental requirement for enforcing bandwidth guarantee in the SDDS system is to first collect the bandwidth requirement from the tenants at the time of creating a VD/VDC. In order to ensure strict enforcement of the bandwidth guarantee, tenants are required to precisely configure the physical-level bandwidth requirement. However, even when the tenants have the knowledge

of the best of the technically mastered system administrators, they fail to ascertain the exact physical-level bandwidth specification, because physical-level bandwidth depends upon a lot of intrinsic factors of an application's data workload like read/write type, request size and sequential locality, and hence the physical-level bandwidth is very different from its corresponding application-level bandwidth. Unfortunately, QoS solutions offered today force the tenants to configure their QoS specification using physical-level bandwidth requirements. Since such a QoS configuration isn't proven to correspond accurately to the application's requirements, even if the SDDS system enforces physical-level bandwidth guarantees accurately at all times, the application performance might not be upto acceptable limits, as expected by the tenant. The SDDS system might enforce the physical-level bandwidth guarantee at all times, but the tenants are forced to accept one of the following options, a) pay a nominal fees to the service provider, but the application performance could deteriorate beyond acceptable limits, b) pay exorbitant fees to the service provider to maintain redundant hardware resources which are used only occasionally, and application performance stays within acceptable limits. Although both the options ensure 100% adherence to the physical-level bandwidth guarantee, it is not in the best interest of the tenant because while a) leads to deteriorated application performance, b) results in an expensive solution. Tenants' applications have a very simple QoS requirement, which is to ensure good performance. But unfortunately the precise meaning attached to the goodness factor is only qualitative. Tenants desire an SDDS system that can ensure both the optimal usage of hardware resources which brings down the total cost, and an efficient mapping between the physical-level bandwidth requirement and the goodness factor associated with an application's performance. Given the current day technologies, such a solution is a far cry from reality. Fortunately, most of the times, tenants do have a way to specify application-level bandwidth requirements, and hence there is a great need for an efficient tool that can automatically translate application-level bandwidth requirements into physical-level bandwidth requirements.

Physical-level bandwidth specification of a VD/VDC is the key factor that controls the accuracy with which the SDDS system enforces QoS guarantees to that VD/VDC and hence it is very important for the SDDS system to capture a comprehensive view of the data locality information of a VD/VDC's disk I/O workload while extracting the physical-level bandwidth requirement. Physical-level bandwidth requirement can be expressed in several different ways, like MBPS, IOPS, or disk resource usage time (DRUT). In order to express physical-level bandwidth requirement in terms of IOPS, it is sufficient to measure the average number of I/O requests in a VD/VDC's workload

that are processed by the corresponding DAs over a small window in time. Though, IOPS captures the sequential/random locality and read/write type of I/O requests in the workload, IOPS alone isn't sufficient to capture the overall locality of the data workload, because it varies considerably for I/O requests of different sizes. For example, a disk with a rating of 200 IOPS and I/O request size of 4 MB will perform better than a disk with 1000 IOPS and I/O request size of 4 KB. Expressing physical-level bandwidth requirement in terms of MBPS solves this problem, because it considers the number of bytes rather than the number of I/O requests that are processed by the DAs in a small window in time. However, both these techniques give no guarantee on the individual latency of an I/O request, though they do guarantee on the average I/O latency over a longer term. A third alternative is to measure, DRUT, which is the amount of time that a DA is actively used to service the I/O requests from different VD's that share that DA. Since DRUT captures the I/O latency of every I/O request in the incoming disk access workload on a DA, it effectively captures the read/write ratio, disk access locality and the I/O request size in the data workload. However, expressing physical-level bandwidth specification using any of these approaches involve significant challenges in a distributed storage environment.

Due to 3-way replication technique adopted by the SDDS system, a stream of disk I/Os from a VD is split into multiple sub-streams targeted to three different DAs located on three different SNs. Therefore, to capture the data locality in a VD's workload, one option is to collect the disk I/O statistics before the disk I/O stream is split into multiple sub-streams. Though, it is relatively straightforward to collect the disk I/O statistics at the source itself, it is impossible to extract accurate locality information in the workload because the main component of the disk I/O statistics is the end-to-end I/O latency of the I/O requests in the workload and the observed I/O latency includes disk I/O latency on the target DA in addition to the network latency associated with communication between a CN and its corresponding SN. Estimation techniques have been proposed by earlier research works [143] to approximate the I/O locality in the workload, but because it isn't completely accurate, either the QoS specifications aren't enforced strictly or the hardware resources aren't utilized optimally. Another option is to collect disk I/O statistics on the destination DAs itself. Though NAS/SAN systems project the storage system like a black-box and makes it impossible to install any customized software on the storage system to collect disk I/O statistics, the JBOJBOD setup makes it possible to collect disk I/O statistics directly inside each SN, on the corresponding DAs. Though, disk I/O latency and other attributes of a disk I/O request can be captured accurately at the destination DA, it is extremely chal-

lenging to correlate the disk I/O statistics from each sub-stream and capture the data locality of a VD's workload, at real-time.

The above mentioned problem is magnified when thousands of VDs and DAs are involved. In such a setup, it is also challenging to identify overloaded servers and to distribute the load evenly among all the DAs. Typical load balancing techniques [144] migrate some workloads from a overloaded DA to an under-loaded DA, until the over-loaded DA is no longer over-loaded, but migration is a very expensive operation, as it involves moving the entire workload between two DAs. An alternate solution is to exploit the nature of read-write disk I/Os in an SDDS system with inter-storage-node data redundancy. Due to strict consistency requirement of the SDDS system, there isn't any flexibility to re-route a write I/O request because a write I/O request has to be submitted to all replicas synchronously. However, a read I/O request can be submitted to only the least loaded DA among the replicas, and hence the selection policy to choose a least loaded DA serves as a control knob to balance the load across the DAs. However in a large-scale distributed setup like SDDS system, it is not straight-forward to choose a least loaded DA at real-time. If a VD's scheduler routes its read I/O requests using localized decisions within that VD, then it is possible that multiple VDs that share a DA, can each make best localized decisions within the VD, yet they might result in a mixture of few overloaded and underloaded DAs. Therefore, at all points in time, every VD's scheduler has to guarantee to submit its read I/O requests to the least loaded DA among its replicas, such that every VD shares the same view of the overall load on any DA. However there are multiple challenges in ensuring such a guarantee. First, every VD needs an accurate mechanism to understand the load on all its replicated DAs before submitting its read I/O request to the least loaded DA. It is impractical to get such information before submitting every read I/O request because the DAs are remotely located from the VDs and hence could incur large latencies. Second, a DA services requests from multiple VDs that are located on different CNs, and hence its impractical for a VD to communicate with all such VDs before making every routing decision. Third, the latency associated with a given read I/O request includes network latency and the I/O latency experienced on the DA. Both these latencies are subject to large variations due to multiple factors like request size, locality, read/write type and request rate in the input workload. Its difficult to isolate these components of an I/O latency and hence a VD with localized scheduler finds it difficult to analyze the load on a DA. Therefore, due to lack of coordination between the VDs and lack of centralized management, read load balancing is a challenging problem in the SDDS system.

The read load balancing issue mentioned above is further complicated, because it is not sufficient if the read I/O requests in a VD are distributed to globally balance the load on all the DAs in the SDDS system. It is equally important to maximize the data access locality in a VD's data workload, whenever possible. Therefore, the local load balancer on a VD has to carefully ensure long-term global load balance on all the DAs, while simultaneously ensuring short-term gains in maximizing data access locality.

It is incorrect to assume uniform data locality in a disk I/O workload at all points in time. In the event of bursts or short-term fluctuations in either the workload locality or the rate of data flow in a workload, the SNs should adopt suitable flow control mechanisms to instruct the storage clients (CNs) to throttle the data flow rate between the given VD and DA. In a SDDS system with different QoS priorities assigned to each VD, the flow control mechanism has to assign the data flow rate to VDs in proportion to their bandwidth guarantees, but that requires accurate measurement of the instantaneous residual bandwidth of a DA and its a key challenge to the flow control management in a distributed storage system.

While read load balancing and flow control techniques avoid overloaded DAs, care should be taken to ensure that the DAs are not under-utilized too. On a DA, I/O requests from different VDs arrive at unpredictable intervals in time. The latency of an I/O request directly depends upon the latency of the previously processed I/O request on a disk. If consecutive requests submitted to a disk from different VDs are located on different tracks, then the last issued I/O request incurs a rotational and/or seek-latency. It is quite possible that if both the VDs have workloads with high sequential locality, then neither of them would experience the additional latency. We refer to such an undesirable latency as storage virtualization tax and it is quite tricky to decide as to which VD should be accountable for the storage virtualization tax. In spite of advanced caching and NCQ techniques on modern disk drives, and best attempt to ensure performance isolation between the VDs that share a common DA, it is impossible to ensure absolute performance isolation because of the constraint to prioritize QoS enforcement. Its a common approach to use redundant hardware resources to make-up for the storage virtualization tax as a result of performance interference between the VDs, but if hardware resources are to be used optimally, the storage virtualization tax should be shared among the DAs in a fair manner. Therefore it is a challenging task to assign this virtualization tax to the workloads in proportion to the amount of noise they make, while simultaneously ensuring short-term high disk bandwidth utilization and long-term enforcement of bandwidth guarantee.

7.2.3 Solution Overview

In this work, we propose *Cheetah* to accept the QoS specifications as $\langle B, E \rangle$, where B is the application-level bandwidth requirement and E is the elasticity with which application-level bandwidth is guaranteed. In order to help a tenant convert his application-level bandwidth requirement into physical-level bandwidth requirement, *Cheetah* provides a tool that automatically extracts this information from a small sample of the application's workload. Since some tenants might pre deterministically expect a variation in the workload pattern, *Cheetah* enables such tenants to control the bandwidth component of their specification by using the elasticity metric. As an example, a tenant planning to provision the SDDS system for a mail server, could either configure E $>$ 100%, say 200%, to indicate that the application is expected to issue twice the number of emails that are present in the sample workload, or he can configure E $<$ 100%, say 30%, to indicate that its sufficient to guarantee only 30% of the application-level bandwidth that is present in the sample workload.

We describe in section 7.3, how *Cheetah* analyzes a small representative sample of the tenants' application's I/O workload and automatically extracts vital information concerning the data locality in the workload. It quantizes the abstract application-level bandwidth requirement into physical-level bandwidth specification, that is easier to comprehend and enforce on the storage devices. Once *Cheetah* admits a VD/VDC along with its bandwidth reservation, it decomposes the per-VD and per-VDC level QoS reservations into that of the corresponding DAs. In section 7.4, we describe how *Cheetah* uses a centralized read load balancer to periodically collect load information from each DA and then adopts a novel piecemeal iterative load-sensitive VD-DA weighted assignment algorithm to determine the distribution pattern of read I/O requests on each VD, such that none of the DAs are overloaded in the entire storage system. The centralized read load balancer doesn't bottleneck the data flow path, because it runs only in the background at periodic intervals, to suggest distribution of read I/O requests to each VD. Each VD still makes a localized decision to distribute the read I/O requests by considering caching and locality in the input workload over the short-term, and uses the hints from centralized controller to decide on the loading factor of a DA, over the long term. In Section 7.5, we describe how *Cheetah* computes the data flow rate between VDs and DAs in proportion to the corresponding QoS priorities on the VDs. The flow control algorithm is efficiently managed using the accurate representation of physical-level bandwidth specification and residual bandwidth information on each DA. In case of short-term fluctuations, VDs regulate the data flow rate based on the suggested flow control hints from the corresponding DAs. Finally, in section 7.6, we describe how *Cheetah*

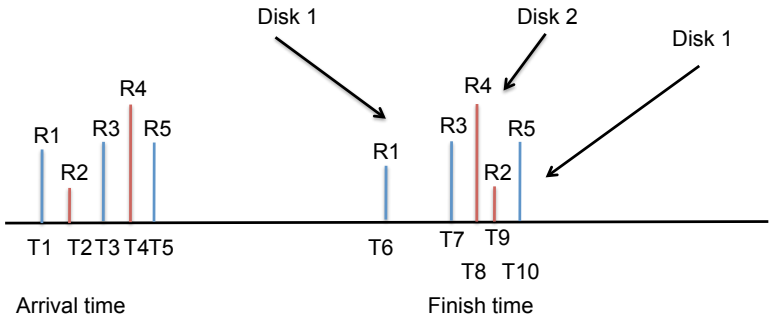

Figure 7.2: *Illustration of DRUT computation on a DA shared by multiple VDs*

uses a QoS aware disk I/O scheduling algorithm, CFVC [135], in each DA to process disk I/O requests from several VDs/VDCs, while ensuring maximum disk bandwidth utilization on the DA, and fairness and performance isolation among the VDs/VDCs that share the DA.

7.3 Quantification of Physical Disk Resource Requirements

In order to convert application-level bandwidth requirement into physical-level bandwidth requirement for a VD, *Cheetah* analyzes a sample of VD's workload for a small time window of T seconds, on a set of dedicated DAs that are specifically meant for sampling purposes. These dedicated set of DAs are used to analyze the sample workload and during this stage, each VD is considered one at a time, because the DA is still not in a position to efficiently handle workloads from multiple VDs simultaneously. To comprehensively capture the locality information of the VD's workload, *Cheetah* expresses physical-level bandwidth in terms of DRUT. To measure the DRUT capacity of a VD, *Cheetah* measures the I/O latency of every disk access request on the DA and aggregates all such latencies to compute the total I/O time. Due to advanced NCQ and caching techniques on the DA, some I/O requests are overlapped and hence *Cheetah* identifies and filters out the overlapped portions of the I/O latencies from the total I/O time to calculate the total effective duration for which the DA is actively used to service the requests from the given VD, which is the DRUT capacity of that VD. Since DRUT measures the effective disk usage time, it inherently captures all the core variables of a data workload like read/write ratio, request size and the amount of randomness (locality), which is why *Cheetah* prefers to express the physical-level bandwidth requirement of a VD in terms of DRUT rather than IOPS, MBPS or other similar

terminologies. Later, in the performance evaluation section 7.9.1, we show the DRUT capacity for several real-world applications like web-search engines and online transaction processing systems, and show that the DRUT metric indeed captures all the important locality information in the I/O workload.

Figure 7.2 illustrates how *Cheetah* computes the DRUT capacity of a VD in a DA. The DA receives I/O requests R1, R2, R3, R4 and R5 from VD1 at times T1, T2, T3, T4 and T5 respectively. In this example, we assume the DA is configured with RAID0 setup with 2 disks and we also assume that the incoming requests are sequential in nature. Therefore, alternate requests are serviced by the same disk in the DA. Due to advanced disk scheduling techniques, requests are merged and reordered as shown in the figure. R3 and R5 are merged together and hence their interrupts are coalesced. Similarly R2 and R4 are processed together by the DA, and R1 is processed alone and could not be merged with R3 and R5 because it probably arrived a bit earlier than the threshold time for merging on disk 1 on the DA. *Cheetah* computes the average I/O latency for each group of I/O requests that are processed together at the same time (whose interrupts are coalesced). The I/O latency for R1, and the average I/O latency for the group of R3 and R6 are aggregated together to compute the DRUT capacity for disk 1. Similarly the average I/O latency for the group of R2 and R4 are aggregated together to compute the DRUT capacity for disk 2. *Cheetah* computes the average I/O latency for a group of requests in a very careful manner, such that the latency should not overlap in time. Therefore, for the group of R3 and R5, *Cheetah* computes the difference between T10 (the time at which last request in the group received its interrupt acknowledgement) and T6(the time at which the first request in the group began to be processed), and assigns the average of this difference as the I/O latency for each of the requests R3 and R5. If T3 happens to appear after T6, then the start time of the group is considered to be T3. *Cheetah* aggregates the average I/O latency for each I/O request in a VD on every disk in the DA to determine the DRUT capacity for each disk in the DA. *Cheetah* again aggregates the DRUT capacities on each disk for a VD to determine the overall DRUT capacity for a VD on that DA. It should be noted that the average I/O latency associated with each I/O request in a VD is used only for DRUT computation and it doesn't correspond to the actual end-to-end I/O latency of the request on its DA.

It is extremely hard for *Cheetah* to measure the DRUT capacity of a DA configured with a checksum-based RAID setup because it is quite tricky to identify and associate the latency of an I/O request that corresponds to the checksum. We discuss this problem in greater detail in Section 7.6.2, and in

this work we restrict the DA to a software RAID0 or similar stripe-based RAID setup.

During the sampling phase of T seconds, *Cheetah* computes the DRUT capacity of the given VD and uses it to compute the physical-level bandwidth factor (PBF), $PBF = DRUT/T$. Since the I/O latency is measured on the DA, it doesn't take into account the network latency and other queuing latencies involved at other components in the entire SDDS system, and hence the I/O latency effectively captures the core characteristics of the workload in terms of read/write ratio, I/O request size and locality (randomness). The calculated PBF corresponds to a factor of DA's raw disk bandwidth (RB) thats actively used, and RB is calculated as the maximum disk I/O bandwidth available on the DA when the workload consists of only write I/O requests with 100% sequential locality. RB is measured in units of MBPS.The physical-level bandwidth (PB) corresponding to the application-level bandwidth (AB) is calculated using the formula, $PB = E * PBF * RB$, where E is the elasticity configured by the tenant in the QoS specification. The rationale behind including E factor in this formula is because of the following implicit assumptions: a) the core characteristics of the sample workload will continue to approximately remain the same as in the application's real-time workload, except for the arrival rate of the I/O requests, and b) the number of I/O requests are proportional to the number of application requests. The PB value thus calculated, effectively represents the sample workload in its entirety and the entire process is completely automated without requiring the tenants to explicitly specify PB requirements. This PB value by itself doesn't make much sense but when its incorporated into a QoS aware disk I/O scheduler, the requests from different VDs can be processed on a DA with different priorities that are in accordance with the bandwidth guarantees configured in the QoS specification for the respective VDs.

The above mentioned PB extraction procedure focusses only on VD-level QoS granularity, but it is trivial to extend it to VDC-level granularity. At VDC granularity, a tenant's application is spread across multiple VDs located on physically isolated CNs. Given a VDC, the tenant's application workload is sampled on all the VDs belonging to the VDC as previously explained, but with just a minor exception. The CFVC scheduler in each DA would maintain a queue for each VDC rather than for each VD. Therefore, *Cheetah* aggregates a set of PB values from all the DAs that hold the VDs belonging to the given VDC and uses the aggregated PB value as the physical-level bandwidth to be guaranteed for that VDC. Since the CFVC scheduler in a DA aggregates the I/O requests from all VDs belonging to a VDC, temporary fluctuations in one of the VDs of a VDC is efficiently absorbed in the corresponding DA. However,

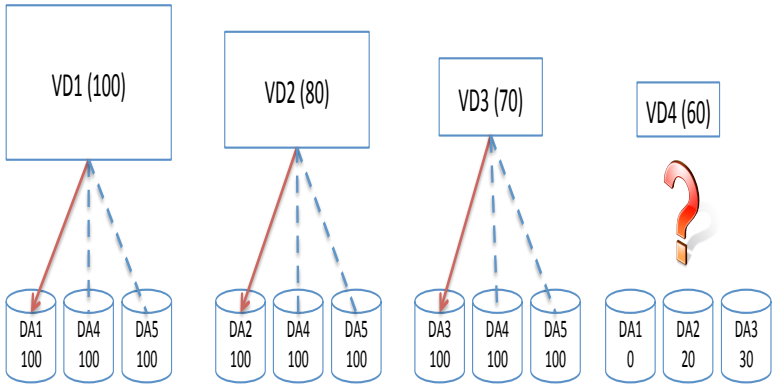

Figure 7.3: *Illustration of a scenario where a naive greedy RLB algorithm fails to load balance the DAs*

if the VDs are located in different DAs, *Cheetah* needs to build additional non-trivial optimizations, like coordinating between such DAs at real-time, in order to handle such short term fluctuations in the workload, and we reserve it to future work.

7.4 Read Load Balancing

Cheetah uses a centralized read load balancer (RLB) that gathers input workload information of every VD, and the overall disk resource utilization information of every DA, to generate a map of weighted routes between a VD and its corresponding replica DAs. Each VD periodically queries the RLB to use the route map to identify the optimal routes, such that no DA is overloaded. The input workload information of a VD can represent the workload's overall I/O latency, where the overall I/O latency is defined as the sum of disk I/O latencies experienced by the I/O requests in that VD's workload. Both read and write I/O requests are considered while gathering a VD's load on a DA because even though RLB technique is employed to route read I/O requests, the load on a DA is determined by the load exerted by all types of I/O requests on the DA.

In design, RLB collects the load information of every VD and distributes the load in each VD to its corresponding set of N replicated DAs, such that the overall load across all DAs are uniformly distributed. In other words, the RLB scheduler assigns an optimal weight to each route between the VDs and their corresponding DAs, and a local scheduler in each VD routes its I/O requests to target DAs in proportion to the weights thus obtained. Weights on each route depend both on the locality of the workload in the VDs and

on the load on target DAs. Unfortunately the task of finding an optimal weight assignment to each route between x VDs and y DAs in the overall system is quite complicated and since the RLB scheduler works in the scale of hundreds to thousands of DAs and VDs, the time required to find optimal weight assignment is critical to the overall performance of *Cheetah*. If RLB were to apply a greedy approximation algorithm, it would order the VDs in the descending order of the workload's overall latency, and for each VD, the input workload would be split into multiple parts such that each part would be assigned to each of its replicated DAs in proportion to the load on that DA. Since the algorithm begins with the VD with highest load, DAs are expected to be assigned with heavy loads initially and later on with lighter loads. Such an assignment works on the best effort basis to load balance the DAs, but it is quite possible for a VD to face a situation wherein it has to select a least loaded DA such that the selected DA is already overloaded due to some earlier assignments. Ideally, using a dynamic programming solution at such a stage would lead to backtracking the assignment process until atleast one of the target DAs for a VD is not overloaded. But due to time constraints on the RLB to find a weight distribution solution at real-time, the greedy algorithm would be forced to select an overloaded DA as the least loaded DA among the possible alternatives for a VD, and hence such an unbalanced load distribution forces *Cheetah* to fail in ensuring the QoS guarantees. Figure 7.3 illustrates the load balancing problem using 4 VDs and 5 DAs. The naive greedy approximation algorithm maps VD1 to DA1, VD2 to DA2 and VD3 to DA3, but when it comes to VD4 with a load of 60 units, none of its replica DAs (DA1, DA2 and DA3) have enough capacity to hold its load. Therefore, VD4 is forcefully assigned to the least loaded DA (DA3) among the available options, and clearly DA3 will fail to serve the collective load of 130 units using its capacity of 100 units.

This optimal weight assignment problem is similar to the problem that appears in network switch scheduling, where the switch needs to find a way to maximally pair a set of input ports with a set of output ports. Fortunately, in the network switch scheduling literature, PIM [160] and iSLIP [161] suggest a greedy approximation algorithm using multiple iteration technique to answer this issue with a practically efficient solution. *Cheetah* adopts this multiple iteration technique to enable RLB to split the input workload into a number of load units that are much greater than the number of replica choices available for a VD and then iterates multiple times to distribute the load units among target DAs. Since the load assignment for a VD is done over multiple smaller installments, the proposed greedy algorithm significantly reduces the probability of selecting an overloaded DA as the least loaded DA option for

any VD. However, the multiple iteration technique doesn't completely avoid the possibility of having an unbalanced load distribution. The size of the load unit in each VD is critical to this entire load balancing operation. While a large load unit increases the possibility of imbalanced DAs, a smaller load unit decreases the possibility of imbalanced DAs at the expense of increased computation time, running over large number of iterations. Hence RLB computes the size of the load unit to be the mean of the disk channel utilization time of every VDs workload.

Cheetah collects the input workload information from the DAs rather than the VDs because of two reasons. First, the number of VDs far exceed the number of DAs and hence gathering statistical information from each DA helps save the network bandwidth. Second, the disk channel utilization time information measured on a DA implicitly covers the important variables like locality in the workload, I/O request size, read/write type and request rate, that would otherwise pose severe challenges if these variables were to be extracted individually based on the end-to-end latency as observed from a VD.

7.4.1 RLB Algorithm

RLB requires the following input information: a) Input workload information of all VDs in terms of disk resource utilization time of the VDs workload, b) Capacity of each DA, in terms of maximum IOPS supported by the DA and c) Average I/O latency of the requests processed by every DA. The capacity of a DA is computed as the number of fixed-sized sequentially processed I/O requests per second and its just a relative measure to fairly identify DAs with different processing capacities. The average I/O latency of a DA is computed by taking the average of the latencies of I/O requests from all VDs that are processed by that DA.

RLB splits the overall load on each VD into multiple fixed size load units, such that ith VD has K_i load units. As the output of the RLB algorithm, RLB assigns each VD with a *local weight* to each of its replicated DAs, and it is these local weights that the a VD uses as a heuristic to distribute its read I/O requests among its replicated DAs. Therefore RLB maintains a counter to record local weight for each replica DA of a VD. Additionally, RLB also maintains a global counter for each DA to record the global load on that DA as a result of the load assignments for a VD-DA pair.

Load units are assigned from VDs to their target replica DAs using the following algorithm:

1. Sort the x VDs in descending order of K_i values to produce a sequence VD1, VD2, ... VD_x

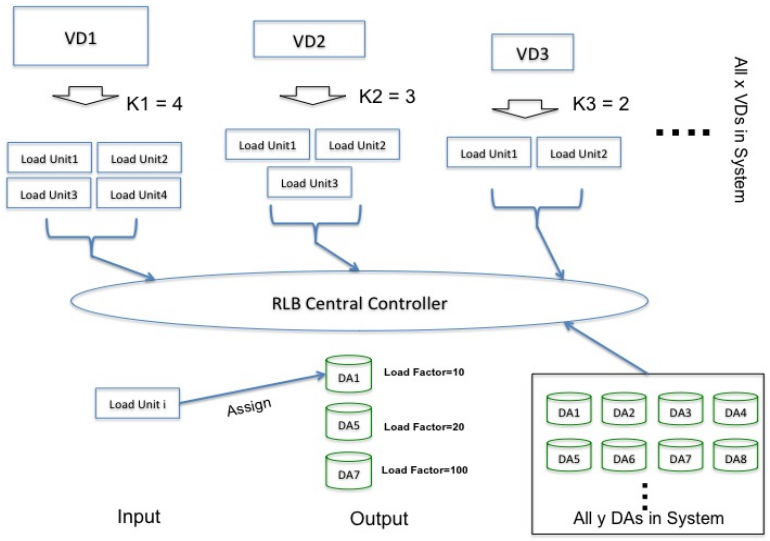

Figure 7.4: *Illustration of RLB management. All x VDs are arranged in decreasing order of their K_i values. Number of replicas are shown to be 3. For every load unit, it's corresponding 3 replica DAs out of the overall y DAs are identified and arranged in increasing order of their load factor. In every iteration, one load unit is assigned to the least loaded DA (lowest load factor).*

2. For the first VD in the ordered list, select the target DA as the DA with least load factor among the VD's replica DAs. Assign a load unit from the VD to that target DA by incrementing the local weight for the target DA in that VD by one, and also increment the global load counter for that DA by a value equalling to the average I/O latency on that DA. A DA is said to have the least load factor if it has the least ratio of $\frac{global\ load\ counter}{Total\ Capacity}$ among all the replica DAs corresponding to that VD.

3. Insert the previously selected VD in its correct place in the ordered list. If the VD doesn't contain any more load units, then remove it from the list.

4. Continue the steps 2 and 3 until the list is empty.

Figure 7.4 illustrates the above algorithm. The key ideas in this greedy heuristic algorithm are: a) to do each VD's load assignment to its replica DAs in multiple smaller installments so as to prevent the greedy assignment from making too big a mistake, b) to allow each VD a number of shots in assigning its load units to its DAs, where the number of shots is proportional to its load, c) to assign the load units to DAs in sequence starting from most loaded VDs till the least loaded VDs, where assigning a load unit to a DA is

equivalent to submitting a group of read I/O requests to that DA. Since each DA is shared by different set of VDs, the average I/O latency experienced by a group of read I/O requests is different on each replica DA. Therefore in step 2, upon selecting the DA with least load factor among the replica DAs, the global weight for that DA is incremented by a factor equalling to the average I/O latency observed on that DA, rather than by a fixed global value shared among all he DAs. The least loaded DA selection process involves the capacity of a DA, because in a distributed storage system it is quite common to see DAs with a different capacity and it is necessary to assign more load to DAs with higher capacity.

7.4.2 RLB Integration with *Cheetah*

RLB algorithm is integrated into *Cheetah* as follows:

1. For every sampling window of T minutes, each DA gathers the following information about the processed I/O requests for each of its VD, a) cumulative I/O latency b) total number of requests. The sampling window size should be chosen large enough to capture a consistent overall workload pattern in each VD and through our empirical observations we suggest a value between 1 to 3 minutes for the sampling window T.

2. At the end of the sampling window time, each DA submits to RLB the average I/O latency, total request count for each of its VD and the total capacity of that DA in terms of maximum IOPS.

3. RLB waits for a small predetermined time after the sampling window, to ensure all DAs have submitted their observed results and then runs the RLB algorithm to calculate local weights for all the replica DAs in each VD.

4. Each CN queries RLB at periodic intervals to collect the DA load distribution map for each of its VDs.

5. Each VD uses this load distribution map to accordingly distribute the read I/O requests among its replicated DAs.

Steps 1-4 described above, contribute to generating a load distribution map that is then passed on to each VD to use as a heuristic to make efficient read I/O request routing decisions as described in step 5. While, steps 1-4 make a best effort to ensure that the overall storage system is uniformly loaded, step 5 ultimately decides the optimal route for each read I/O request in a VD. Thus, in step 5, the local scheduler that makes decisions to route the read requests is

very critical to the entire load balancing process. A naive local scheduler that blindly distributes the read I/O requests in strict accordance to the heuristics in load distribution map will ensure uniform load distribution on all the DAs in the system, but it might not be in the best interest of the system's overall performance. There are a couple of important points that deserve a detailed analysis. First, the latency aspect of the QoS guarantee is not addressed in this work. In order to ensure latency guarantees in the QoS, the local scheduler in a VD needs to route every read I/O request to the best replica DA that helps deliver lowest I/O latency and selecting such a DA could temporarily violate the ideal load distribution heuristics suggested by the RLB algorithm. There could be several reasons for such a routing decision and one of them is to optimally utilize the DA cache using the locality information in the incoming workload on a VD. Second, it is not in the best interest to ensure uniform load on all the DAs at all times. Sometimes it is beneficial to overload a DA and under load another DA by redirecting a majority of the requests from a VD to a single DA, because each DA could then have lesser INTER-VD seeks to do. In the QoS aware disk I/O scheduler in a DA, it is clear that whenever a request has to be handled from a different VD, a large seek is performed and that results in a very high I/O latency. The above two points require additional research to guarantee optimal overall system performance and hence we leave these two concerns for future research. However, in this dissertation, we build a simple local scheduler in each VD that incorporates both the VD's workload locality and the RLB weight distribution map. The local scheduler keeps a LRU cache of the logical block addresses (LBA) of X read I/O requests along with the target DAs to which those requests were submitted. Assuming that a DA can use its disk cache better when the incoming I/O requests are sequentially located, the local scheduler on a VD chooses a replica DA to submit a read I/O request, R1, using the following 2 steps:

1. Identify an I/O request, R2, whose LBA is closest to R1 and if the difference between the LBAs of R2 and R1 is less than T sectors, then choose the target candidate DA as the DA corresponding to R2. T is chosen such that there is a high probability of processing R1 from disk cache on the target DA or atleast if R2 and R1 are located on the same or adjacent tracks on the disk. Since it is extremely difficult and undesirable to expose disk level semantics to each VD, based on empirical observations, T can be roughly approximated to a value around 20K.

2. If the candidate DA chosen in step 1) doesn't violate the RLB weight distribution, then that DA is chosen to submit R1, but if it does violate, then a DA that doesn't violate the RLB weight distribution is chosen to submit R1.

Therefore, the local scheduler on the VD makes locality-aware decisions while distributing the read I/O requests, and hence ensures uniform load balancing on the DAs, together with minimal I/O latency for each I/O request and effective DA bandwidth utilization.

7.5 Flow Control

The bandwidth guarantee enforced by *Cheetah* only ensures the minimum level of service for each VD/VDC, but on a distributed storage system shared by multiple tenants, it is quite natural to see some VDs generating more workload that seeks a higher share of the DA than what it had requested for in its QoS specification. If the DA does not have spare bandwidth, then such a workload overloads the DA and can potentially bring down the performance of the entire DA. Though the CFVC scheduler in the DA ensures fairness and performance isolation to a fair degree, it is beyond the control of any disk scheduler to stop a workload from generating more requests. Since the hardware resources on a SN are limited, it is essential to regulate the data flow between CNs and SNs, so that a CN never overloads any SN. When a SN suggests some data flow rate to a CN, the CN has to again adopt another flow control algorithm to regulate the data flow from multiple VDs that send their data to multiple DAs on the given SN. Otherwise, a single VD that misuses its share, will negatively impact other VDs on that CN that share the same SN. Therefore, *Cheetah* proposes to regulate the data flow rate directly between DAs and VDs. An SN computes the ideal data flow rate for each of its DAs and forwards the flow control suggestions to the corresponding CNs that hold the corresponding VDs.

To handle flow control management between VDs and DAs, it is necessary to accurately measure the residual bandwidth on each DA in the storage system and only then the SNs can suggest a suitable data flow rate to the corresponding VDs. However, due to advanced caching and NCQ techniques on modern disk drives, it is extremely challenging to measure the DRUT capacity of a VD, which was described in greater detail earlier in Section 7.3. Given the DRUT capacity for each VD on a DA, *Cheetah* uses the following flow control algorithm on each of the DAs:

1. Measures the total disk time usage Y, and the disk time usage Y_i and the incoming I/O rate Z_i of each VD_i that imposes load on that DA, and

2. Sends to VD_i an advised I/O rate, which is equal to

$$\left(\frac{Y_i}{Y} * Y_{limit} * \frac{1}{Y_i} * Z_i\right) = \left(\frac{Y_{limit}}{Y} * Z_i\right)$$

Y_{limit} is each DA's maximum allowed disk time usage. If Y exceeds Y_{limit}, flow control is triggered. The advised I/O rate given by a DA to a VD sets an upper bound on the I/O rate of that VD to that DA.

Y_{limit} is configured as a percentage of the time period for which the statistical measurements are made on the DA. Very low value of Y_{limit} results in under utilization of disk resources and a very high value results in overloaded conditions, that seriously disrupts the overall performance of the system. Therefore, as a safe heuristic, *Cheetah* configures Y_{limit} as 70% of the total observation time. In the formula used in step 2, when Y exceeds Y_{limit}, for each VD_i, Z_i should be lowered by a factor:

$$\frac{required\ disk\ usage\ time}{observed\ disk\ usage\ time}$$

where, required disk usage time should be a factor of Y_{limit} rather than Y, and hence the factor $\left(\frac{Y_i}{Y} * Y_{limit}\right)$.

In the above mentioned flow control algorithm, all VDs are treated equally and hence is QoS unaware. The required disk usage time component in the above mentioned QoS unaware flow control algorithm, should ideally involve QoS reservations and hence *Cheetah* proposes the following QoS aware flow control algorithm, and does the following on each DA:

- Measures the advised I/O rate to be

$$\left(\frac{QoS_i}{QoS_{sum}} * Y_{limit} * \frac{1}{Y_i} * Z_i\right)$$

QoS_i is the bandwidth requirement of VD_i on the DA and QoS_{sum} is the sum of all QoS requirements on that DA.

In case of high fluctuations in input workload pattern, the observed disk usage time gives a tighter control over the flow control regulation rather than the QoS aware formula, and hence *Cheetah* computes the final advised I/O rate as the minimum of,

$$\left[\left(\frac{QoS_i}{QoS_{sum}} * Y_{limit} * \frac{1}{Y_i} * Z_i\right), \left(\frac{Y_{limit}}{Y} * Z_i\right)\right]$$

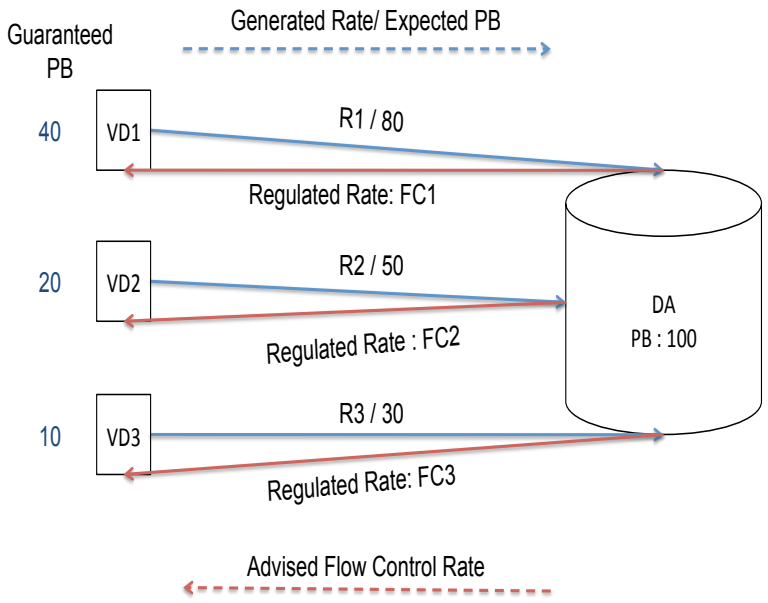

Figure 7.5: *Illustration of flow control management on a DA shared by 3 VDs. PB is in units of MBPS*

Figure 7.5 illustrates an example of flow control management on a DA shared by 3 VDs. The PB guarantee for VD1, VD2 and VD3 are 40 MBPS, 20 MBPS and 10 MBPS respectively. The DA with 100 MBPS PB capacity is ideally configured to be 70% utilized and is reflected by the sum of the guaranteed bandwidth for each VD on the DA, which is 70 MBPS. Lets assume VD1, VD2 and VD3 generates its requests at a rate of R1, R2 and R3 I/O requests/second respectively. R1, R2 and R3 corresponds to hypothetical PB value of 80 MBPS, 50 MBPS and 30 MBPS respectively. The expected PB bandwidth is shown to give a better clarity on the load exerted by each VD on the DA. Since the sum of the expected PB values from all the VDs (160 MBPS) is higher than the desired PB utilization capacity of the DA (70% of 100 MBPS), the CFVC scheduler on the DA proportionally allocates the DA bandwidth to each of the VDs and hence VD1, VD2 and VD3 receives a PB share of 57, 29 and 14 MBPS respectively. *Cheetah* then triggers flow control mechanism to regulate the data flow from each VD to the DA. In the QoS unaware technique, each VD is asked to reduce its request generation rate by a factor equivalent to $\frac{R_i}{R1+R2+R3}$, where R_i corresponds to R1, R2 or R3 for VD1, VD2 or VD3 respectively. However, the request generation rate by itself doesn't represent the entire workload on a VD. Hence *Cheetah* uses the DRUT capacity of a VD to determine the factor by which the VDs request generation rate should be reduced. The DRUT capacity of a VD effectively captures

all the necessary locality information of a VD's workload and corresponds to the PB value allocated by the CFVC scheduler. Similarly, in the QoS aware technique, *Cheetah* uses the guaranteed PB value to determine the factor by which the request generation rate of a VD should be reduced.

7.6 CFVC: A QoS Aware Disk Scheduler

Each DA receives disk access requests from different VDs and for each VD, *Cheetah* is responsible for ensuring strict adherence to the QoS guarantees made to that VD. On a DA, the disk I/O requests from different VDs have no common locality and they arrive at totally unpredictable intervals. Hence it is a non-trivial task to schedule the I/O requests from different VDs in the right order such that the following constraints are met successfully: a) PB value for each VD is honored, b) the raw disk bandwidth of each DA is optimally used, and c) the performance of each VD is isolated, such that a noisy VD generating excessive workload doesn't disturb the performance of a well behaved VD that generates workload in accordance to expected limits. These I/O scheduling restrictions are similar to those of a disk scheduler in a directly attached disk, where the VDs correspond to different processes with different priorities. Such disk scheduling algorithms are well studied in the literature and hence we wish to adopt a well proven QoS aware disk I/O scheduler into *Cheetah* and provision each DA locally with such a QoS aware disk I/O scheduler.

While scheduling algorithms like SCAN [194], CSCAN [195] and weighted round robin [162] do not bound the worst case latency, fair queuing (FQ) [196], weighted fair queuing (WFQ) [197], virtual clock (VC) [198] and CSCAN based virtual clock (CVC) [136] algorithms do not ensure fairness in a short term period. CSCAN based Fair Virtual Clock (CFVC) [135] scheduler not only promises to ensure short term fairness, but also provides performance isolation as well as other features supported in a standard VC scheduler. Therefore *Cheetah* adopts CFVC disk scheduler in every DA.

7.6.1 CFVC Scheduler Algorithm

CFVC scheduler maintains two queues to aggregate disk access requests from all the VDs and periodically dispatches a request to the DA by picking the best candidate from one of the queues. The two queues are, *utilization queue in which the requests are ordered in standard CSCAN order [195], and* QoS queue in which the requests are ordered by their finish time as determined by the virtual disk switching overhead (VDSO) algorithm. The CSCAN and

VDSO algorithms are described shortly. For each incoming disk access request in a DA, the CFVC scheduler calculates the finish time and inserts it in both the queues according to the order maintained in the respective queues. Simultaneously in the dequeue process, the CFVC scheduler checks if servicing the head request in the utilization queue will violate the finish time of the head request in the QoS queue and dispatches the head request in utilization queue if it won't; otherwise dispatches the head request from QoS queue. By giving first priority to utilization queue, maximum disk bandwidth efficiency is extracted from the DA and by ensuring the finish times in QoS queue are not violated, fairness is guaranteed among all the VDs.

CSCAN algorithm orders the requests in utilization queue in increasing order of logical block addresses within a DA. Once the disk head reaches the end of the DA (largest logical block address in the queue), it makes a big jump to the start of the DA and because it continually sweeps across the DA, seek latency and rotational latencies are avoided to the maximum possible extent, thereby guaranteeing maximum possible raw disk bandwidth in DA.

VDSO algorithm computes the finish time for each disk access request using the following formula:

$$FT(r_j^i) = max(AT(r_j^i), FT(r_j^{i-1})) + \frac{L(r_j^i) + VDSO_j * B_{raw}}{B_j} \tag{7.1}$$

Where, jth VD's bandwidth is B_j and its ith request is r_j^i. L() denotes the length of the request's length. Arrival time (AT) is the current wall clock time on the SN server at which a request arrives on the SN. Finish time (FT) is also based on current wall clock time on SN server and it is guaranteed by CFVC scheduler to submit the request to disk within it's finish time. B_{raw} indicates the raw transfer bandwidth of the DA and that corresponds to the 100% sequential bandwidth as explained in section 7.3. $VDSO_j$ represents the per disk access overhead associated with the jth VD and is calculated as described in the following algorithm:

Algorithm 1 Algorithm to calculate VDSO

INPUT:
VD of current request: N;
Current request's disk service time: svr;
Target VD of previous request: prevVD;
Difference in LBN of current request and the previous request on the same VD: LBNGap;
Average Inherent Disk Access Overhead of all VDs, where for nth VD it is: $AIAOH_n$;

OUTPUT:
Virtual Disk Switching Overhead: VDSO;
Inherent Disk Access Overhead of all VDs, where for nth VD it is: $IAOH_n$;

CONSTANTS:
Disk Cache Access Overhead: DCAOH;
LBN gap threshold: LBNGapThreshold;

```
1: loop
2:     if svr <= DCAOH then
3:         ignore svr;
4:     else
5:         if N = prevVD then
6:             charge svr to IOAH_n;
7:         else
8:             if LBNGap <= LBNGapThreshold then
9:                 charge svr to VDSO;
10:            else
11:                charge AIAOH_n to IOAH_n;
12:                charge svr − AIAOH_n to VDSO;
13:            end if
14:        end if
15:        Assign N to prevVD;
16:    end if
17: end loop
```

When multiple VDs are hosted on a single DA, the overhead associated with moving the disk head to handle requests from one VD to another, is the storage virtualization tax and the VDSO algorithm described in 1 distributes this tax to all the VDs in an efficient way, ensuring overall fairness among the

VDs and maximum raw disk access efficiency in the DA. Inherent access overhead (IAOH) is the overhead associated with a virtual disk when two requests belonging to the same VD incurs disk head movement. This corresponds to the disk access overhead of a VD if it were to be hosted on a dedicated DA. The average inherent disk access overhead (AIAOH), is IAOH divided by the number of requests contributing to the calculation of IOAH. In order to measure the disk service time of a request, a dedicated DA is used for the sampling purpose. For each request, the service time, svr, taken to process it on the DA is measured and used to distribute it to VDSO according to the above algorithm. DCAOH is used as a measure to find if a request could hit in a cache or not. As a close approximation, it is set to the average of the disk service time on a DA, excluding the data transfer time to process a large number of sequential requests. The resulting VDSO from the algorithm represents the overall storage virtualization tax and should be distributed to all VDs in proportion of their IAOHs. Hence on a DA with N VDs, VDSO for each VD is calculated as:

$$VDSO_i = VDSO * \frac{IAOH_i}{\sum_{j=1}^{n} IAOH_j}$$

In order to statistically compute VDSO for each VD, a sample of the workload requests are considered periodically and the corresponding VDSOs are extracted. By periodically sampling the workload, any major changes in the workload pattern is absorbed in the VDSO factor and hence results in a fair and efficient scheduling policy.

The resulting FT in equation 7.1, ensures that for a VD with higher bandwidth reservation B_j, the difference in FTs for consecutive disk access requests is smaller and hence the scheduler will service more number of requests for the VD as expected. However, it may so happen that some of the VDs in a DA are temporarily inactive (backlogged) and one active VD generates requests higher than its reserved bandwidth. The slack in bandwidth from backlogged VDs is efficiently distributed to the active VD and when the backlogged VDs turn active, the previously active VD is paused and the backlogged VDs are serviced until the overall disk channel usage time is balanced for all the VDs in proportion to their bandwidth reservations. However, a typical VC scheduler like this suffers from a short term unfairness problem. During the period when an active VD is paused to serve requests from backlogged VDs, the requests in paused active VD experiences large delays and the delay time is proportional to the amount of slack previously utilized by that VD. Though this technique ensures overall fairness among all VDS by pausing the over serviced VD until the backlogged VDs are given a fair amount of disk channel time, the large delays experienced by requests in over serviced VDs poses an ugly situation,

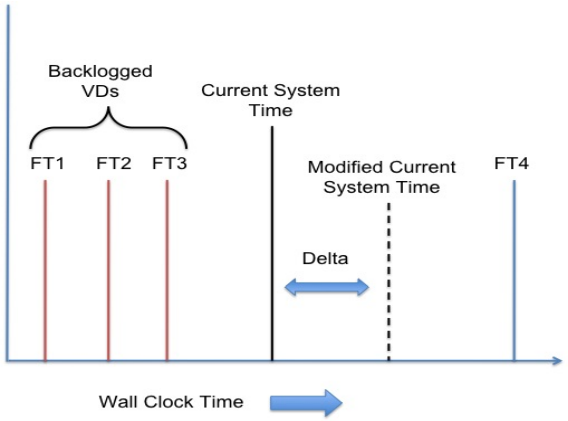

Figure 7.6: *Illustration of short term unfairness problem in a typical VC disk scheduler algorithm. FT1, FT2, FT3 corresponds to finish times of backlogged VDs VD1, VD2 and VD3 respectively. FT4 corresponds to finish time of active, over used VD4. VD4 has used slacks from bandwidths of VD1, VD2 and VD3 due to their temporary inactiveness and hence its finish time is much ahead of the current system time. CFVC algorithm pushes the current system time by delta so that requests from VD4 doesn't wait for long to be processed.*

especially for real time systems. Figure 7.6 gives an illustration of the short term unfairness problem.

The short term unfairness issue is as a result of using wall clock time in calculating arrival time (AT) in equation 7.1. If the current time is pushed ahead by a marginal amount "delta", then for some "alpha" number of requests, the max(...) component in the equation 7.1 results in AT rather than its FT. That ensures those "alpha" number of requests are processed much earlier compared to the situation when "delta" is 0. Thus by controlling delta, the worst case latency can be controlled at the expense of being biased towards the over used VD, where the amount of bias directly corresponds to "alpha", which in turn is controlled by "delta".

7.6.2 Implementation Challenges Integrating CFVC Scheduler into *Cheetah*

Cheetah uses a separate CFVC scheduler for each DA inside each SN. Each SN has a receiver thread that receives incoming I/O requests on a dedicated TCP/IP socket and depending on the target DA, the SN receiver thread forwards the I/O request to a dedicated DA receiver thread for the target DA. The DA receiver thread inserts the received I/O request into a dual FIFO queue maintained by the CFVC scheduler on that DA. Each DA also has a set

of submit threads in the DA and each of these submit threads independently dequeue a request from the dual FIFO queue and submits the I/O request to the target disk in the DA. There are atleast as many DA submit threads, as the sum of the NCQ values on every disk in the DA, because only when multiple threads submit I/O requests in parallel, NCQ advantages on the disk drive are utilized to the fullest extent. Upon processing the I/O request, the DA submit thread forwards the acknowledgement received from the disk drive to a dedicated DA acknowledgement thread, and then returns to dequeue the next available request in the dual queue. Each DA has a dedicated acknowledgement thread that forwards the acknowledgement of the processed I/O request to the appropriate VD using TCP/IP socket interface. The number of threads in this implementation could be reduced if asynchronous I/O operations are used. However, we reserve it for future work and use the above mentioned implementation because the DA threads are majorly I/O bound and even the performance evaluations didn't suggest any noticeable bottlenecks.

There are a number of challenges in implementing the above CFVC design on a DA. First, in order to exploit parallelization in the DA, software RAID configurations with striping functionality like RAID0, RAID5 or RAID10 is chosen. It is necessary to choose a stripe size that maximizes both the disk I/O bandwidth utilization and parallelization opportunity in the DA. Second, the DA submit threads have to identify the target disk for every I/O request. Third, in order to determine the VDSO and IAOH attributes of an I/O request in the CFVC algorithm, the DA submit threads have to determine the accurate I/O latency for each I/O request. Fourth, accurately identify the I/O latency corresponding to checksums in RAID5 type of setting and make it accountable in the CFVC algorithm.

When the stripe size is configured to be lesser than the I/O request size, the RAID array stripes each incoming I/O request into multiple fragments and submits them to the corresponding disks in that DA. Since this results in locking all three disks at the same time, it minimizes the parallelization opportunities that could simultaneously submit I/Os to individual disks drives. When the stripe size is configured to be a multiple of the I/O request size, each I/O request results in internal fragmentation and hence leads to wastage of disk I/O bandwidth. Therefore, it is ideal to configure the stripe size to exactly match the I/O request size.

The CFVC algorithm can ensure absolute performance isolation and high disk bandwidth utilization, only if the I/O latency information is measured accurately for each VD that shares the DA. Due to caching, native command queuing (NCQ), reordering, merging, interrupt coalescing and other such advanced features in modern disk drives, adjacent I/O requests on a DA are

grouped and acknowledged from the disk drive in one shot and hence it is not straight-forward to isolate I/O requests from different VDs in the merged group and calculate the latency of each I/O request. For example, suppose a DA takes 2 ms to process request R1 from VD1 and 3 ms to process request R2 from VD2, when R1 and R2 are submitted independently at different points in time. Now if R1 and R2 are submitted one after the other with a very short time interval, the DA might process R1 in 2 ms and then process R2 in 3 ms, and then return the acknowledgement for both R1 and R2 at the same time. Such a scenario would make R1's I/O latency(5 ms), which is much larger than its expected latency(2 ms). With such a naive latency measurement technique, the effect of R2's locality is incorporated into R1's I/O latency measurement, and hence the CFVC algorithm will fail to deliver accurate performance results. Therefore, *Cheetah* identifies the overall latency of each such merged group of I/O requests and associates the average I/O latency of that group as the I/O latency of every I/O request in that group. The I/O requests can potentially belong to any VD and hence *Cheetah* correctly associates the I/O latency of each I/O request to its corresponding VD and ensures accurate measurement of the attributes in CFVC algorithm. In order to accurately identify the boundaries of such a group, *Cheetah* measures the time interval between consecutive interrupts from a given disk and if the interval is lesser than or equal to the theoretically calculated time taken to transfer the data to disk platter, it is definite that the given I/O request is aggregated with its previous request. However, if the inter-interrupt time interval is greater than the time taken to transfer the data to disk platter, it is not completely true that the I/O requests are not merged, but it is an extremely unlikely scenario and hence it is OK to consider them as not merged.

One might argue that all the above mentioned disk optimizations should be possible only if the consecutive I/O requests are from a single VD and hence it should not be a problem to consider a group of requests from the same VD as one unit. But it is not completely true because it is not a good idea to assume that multiple VDs sharing the same DA, do not interleave the disk space. For example, a DA shared by 10 VDs might be organized such that, each VD occupies 100 MB of interleaved disk space. *Cheetah* intends to distance itself from the structure and organization of a DA, because a deeper dependency to a DA will only make *Cheetah* more vulnerable to frequent software patches, as and when the hardware resources or the mapping structure of the DA is changed.

Since a DA consists of just a bunch of hard disks which can be configured in many ways, software RAID configurations with striping capability are typically preferred due to the possibility of exploiting I/O parallelism in the DA.

With any stripe-based RAID-level configuration, it is possible to deterministically ascertain the target disk drive in a DA using simple modulo arithmetic on the I/O request's logical block address (LBA). However, checksum-based RAID configurations like RAID5, incurs additional I/O latency associated with checksum management and it is important to associate such additional I/O latency experienced due to checksum handling, to the corresponding I/O requests. Unfortunately, since checksums are generated by the software RAID controller, it is not clear as to when and how the disk drives aggregate the checksum disk I/O requests, and as a result, it is extremely difficult to segregate the checksum latency component of the measured I/O latency. For example, lets consider a case where a VD submits write I/O requests $\langle R1 - R100 \rangle$ on a 4 disk DA arranged in RAID5 manner. Since the checksum uses XOR logic, the RAID controller generates 4 I/O requests for R1, two read I/O requests to fetch R1's data block and the checksum block associated with R1, and two write I/O requests to submit R1's data and checksum blocks. Similarly, every I/O request in the VD's workload generates 4 I/O requests and each disk in the DA buffers, re-orders, merges and coalesces requests independently. Lets say, R1's data block read request is merged with R5 and R12's checksum read requests and R16's data block write request. The RAID controller will not return the acknowledgement for R1, until all four disk I/Os associated with R1 are completed. Hence, by the time the RAID controller returns acknowledgement to R1, several requests are merged with R1 and hence the I/O latency of R1 is distributed among all such merged requests.

A similar problem exists when the DA is provisioned with hardware RAID controllers. Even when the stripe size in a hardware RAID5 setup is chosen to be $\frac{1}{N}$ times the size of an I/O request, where N is the number of disk drives in a DA, the individual disks drives are not necessarily synchronized. As a result, the hardware RAID controller stripes every I/O request into N-1 fragments and submits each fragment and the checksum to the corresponding disk drives, but since the disk drives are not usually synchronized, the RAID controller waits until all disk drives successfully process the requests. This waiting time is unpredictable again because of advanced disk optimization techniques and the presence of multiple I/O requests submitted in parallel to the same DA. Only advanced, expensive hardware RAID controllers ensure disk synchronization, but even then it is difficult to segregate the effect of additional I/O latency due to checksums from the measured I/O latency. Therefore, *Cheetah* doesn't use checksum-based RAID settings, rather it uses either RAID0 or RAID10 configuration.

7.7 Putting it All Together

This section gives a consolidated overview of how *Cheetah* uses all the above mentioned novel techniques to enforce accurate QoS guarantees. For the sake of clarity, we will focus on VD-level QoS granularity. When a tenant wishes to create a VD to handle his application's disk I/O workload, he describes the QoS specification as $\langle B, E \rangle$ and submits a sample disk I/O workload of his application. *Cheetah* uses a set of dedicated spare DAs to analyze the sample workload. When the sample workload submits a disk I/O request to its VD, the local disk I/O scheduler in the corresponding CN's DISCO client submits the request to either all the N replica DAs, in case of a write I/O request, or submits the request to one of the replica DAs in a round robin order, in case of a read I/O request. The CFVC disk I/O scheduler on each DA individually measures the latency of each disk I/O request and at the end of the sampling stage, each of those DAs report the PB value for the VD on that DA. Thus *Cheetah* decomposes the AB value into PB value for each VD-DA pair. *Cheetah* uses a naive admission control algorithm (not described in this work) to select a set of N DAs that can accommodate the given VD's PB requirement. If the QoS specification is too high to be accommodated, *Cheetah* reports suitable failure status to the tenant's application and aborts the VD creation process. If the QoS specification is acceptable, *Cheetah* reports the replica DAs to DISCO client and DISCO server to register the VD-DA mapping for future disk I/O accesses to that VD.

For the real-time disk I/O workload generated from the VD, the local disk I/O scheduler for that VD uses a locality-aware scheduling algorithm, to identify target DAs for the read I/O requests in that workload using hints from RLB to ensure that none of the target DAs are overloaded. The read I/O request is then submitted to the target DA. A write I/O request is submitted to all N replica DAs. The DISCO client in the CN, periodically collects load information of the replica DAs from the RLB for all the VDs in that CN, and the local disk I/O scheduler uses this load information to distribute the read I/O requests from each VD.

When the CFVC scheduler on a DA receives a disk I/O request, it inserts the request into the dual queue. The CFVC scheduler processes the I/O request accordingly and upon the I/O completion, it measures and collects the I/O latency of that request. The DA sends suitable acknowledgement to the corresponding VD. Every DA periodically computes the average I/O latency of all the I/O requests processed on that DA and sends to RLB the computed average I/O latency for each VD that submits its workload to that DA, capacity of that DA which is expressed as the maximum IOPS supported by that DA and the number of I/O requests processed by that DA.

When the RLB receives requests from a CN querying for load distribution maps for each of its VDs. It looks up its hash table to aggregate the load information for each VD-DA pair in the request and returns it to the querying CN. When the RLB receives requests from a DA to submit the load information, it registers the collected statistics accordingly. Periodically, the RLB schedules its algorithm to first identify the load units for every VD and then to distribute the load units evenly to all the corresponding DAs using the proposed RLB scheduling algorithm. RLB stores the load distribution weights in a hash table to quickly access them when the corresponding VD queries for it.

7.8 Evaluation Methodology

In this work, we evaluate the correctness and effectiveness of PB extraction mechanism and RLB algorithm. Though it is advantageous to perform the evaluations in a real physical environment consisting of distributed storage system with SDDS capabilities, it is not always possible to arrange such a massive setup for research purposes. More importantly, it is difficult to isolate undesirable side-effects and focus only on the required components in a massively interconnected SDDS system. Therefore, in this work, we use a smaller prototype and an effective simulated environment that captures all the required elements from a real physical environment that are necessary to stress the evaluation of the proposed models.

7.8.1 Current Prototype

In order to prove the effectiveness of PB extraction process, it is necessary to prove that the variations in PB are correlated with the variations in the input workload locality. For the purpose of this evaluation study, the VDs and CNs in the SDDS system do not necessarily have to be real physical systems, but they are necessary to be involved because the number of buffering components and networking interconnects on the data path between a VD and the target disk in the DA plays an important role in the PB extraction process. Therefore, we simulate them with software artifacts, using simple threads and processes, which is described in detail in Figure 7.7.

VDs and CNs are separate software processes and a set of VDs are manually mapped to a CN, such that VDs communicate to their corresponding CNs through TCP/IP socket interface. Each VD has two threads, a submit thread to submit an I/O request to its replica DAs through their corresponding CNs, and a receiver thread to receive acknowledgements from its CN to indicate the completion status of an I/O request submitted to a DA through that

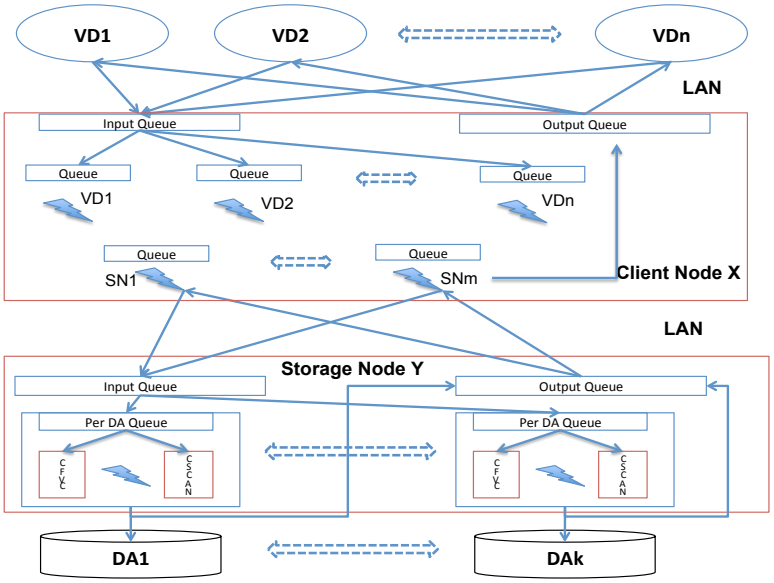

Figure 7.7: *Simulation setup for evaluating the effectiveness of PB extraction*

CN. A VD's submit thread generates requests based on the pre-configured workload setting for that VD. The submit thread also increments a pending request counter to indicate that an acknowledgement is due for the submitted I/O request. Upon a successful TCP/IP transfer the submit thread returns to submit the next request without waiting for the actual I/O completion acknowledgement from the DA, through its corresponding CN.

Each CN process is modeled on the structure of a host operating system in a virtualized system and it contains an input queue, to which multiple VDs submit their I/O requests using TCP/IP socket interface. A dedicated thread in the CN process dequeues from the input queue and inserts it into a FIFO queue corresponding to the VD thats submitted the request. A dedicated per-VD thread dequeues from its FIFO queue, makes the scheduling decision to choose target DAs and forwards the request to the corresponding SN's FIFO queue. A dedicated per-SN thread dequeues the requests from its FIFO queue and transfers the request to the appropriate SN using TCP/IP socket interface.

A SN is a dedicated JBOD storage server, which is a simple x86 system that receives incoming I/O requests on a TCP/IP socket interface and forwards the incoming I/O requests to the corresponding DAs input queue. Each DA is a real physical entity and is represented by a set of dedicated threads on its SN. Each DA has a receiver thread that dequeues from its per-DA incoming FIFO queue, sets a timestamp in the request's payload data to indicate the arrival time on the DA and then inserts the request into the dual queue maintained

by the CFVC scheduler. Each DA also has a set of dedicated CFVC threads for each disk drive in the DA and each of these CFVC threads independently dequeue a request from the dual queue and synchronously submits the I/O request to the target disk in the DA. There are as many dedicated CFVC threads for each disk in the DA as the value of the NCQ on that disk because only when multiple CFVC threads submit I/O requests in parallel, NCQ advantages on the disk drive are utilized to the fullest extent. With a Raid0 configuration, for any given I/O request, it is possible to deterministically ascertain the target disk drive in a DA using simple modulo arithmetic on the I/O request's logical block address (LBA). Upon successful completion of an I/O request, an interrupt is raised by the disk drive controller to notify the corresponding CFVC thread of the I/O completion status. The CFVC thread prepares an acknowledgment request and inserts it into the appropriate output queue on its SN. Each SN has a dedicated acknowledgement thread that dequeues from its output queue and forwards the acknowledgement request to the corresponding CNs.

Each CN has a dedicated receiving thread that receives acknowledgements from the SNs and forwards the requests to the corresponding VDs. The VD receiver thread listens on a dedicated TCP/IP socket and upon receiving an acknowledgement from a DA, through its CN, if the acknowledgement indicates successful I/O transfer, it decrements the pending request counter. Until the number of pending I/O requests in a VD stays below a threshold C, the VD doesn't send any I/O requests but sleeps for a small time period T. The thresholds T (100 ms) and C (10K) are chosen based on simple heuristics to ensure the CPU and Memory resources on the evaluation system are not over burdened.

To demonstrate the effectiveness of the PB extraction process, it is sufficient if a single DA is shown to effectively process I/O requests from a bunch of VDs hosted on a single CN, where the VDs are configured to generate a variety of workload patterns. Hence, the VDs and the corresponding CNs are simulated on a single X86 system with 8 core 2.68GHz processor and 16GB RAM, running 64 bit linux kernel 3.2. Since a DA is just a bunch of disks that are flexible to be configured in different ways, for our experiments we chose to use a software RAID0 configuration using four physical 7200 RPM SATA disks. All the four disks were configured with NCQ value of 31 and the noop disk scheduler was used to avoid undesirable latencies in the linux kernel block layer.

RLB Evaluation Prototype

The presence of CNs and SNs are not strictly necessary to prove the efficiency of RLB algorithm and therefore for RLB evaluation, we have simulated just the VDs and DAs using simple processes and threads, as thats sufficient to stress the RLB algorithm. Each VD process has a dedicated submit thread that submits an I/O request to a DA by inserting the request into a FIFO queue corresponding to the DA. The submit thread then returns to submit its next request without waiting for any acknowledgement. Unlike the above prototype, a real physical DA is not necessary for RLB evaluation. We built a simulated DA that simulates the I/O latency for every I/O request by sleeping for a corresponding amount of time. We describe this simulated DA in greater detail in the next section. The DA simulator processes the I/O requests from the FIFO queue and upon completion, it records the necessary statistics and proceeds with the next request in the FIFO queue. The VDs and the disk simulator collectively use a single X86 system with 8 core 2.68GHz processor and 16GB RAM.

7.8.2 DA Simulator for RLB Evaluation

In order to evaluate RLB algorithms's effectiveness in a cloud-scale storage system, it is necessary to involve several VDs and DAs. However, due to cost restrictions, it is not always possible to evaluate RLB in a real physical environment. It is a challenging task to design a simulated environment and prove its correctness and effectiveness through convincing empirical evaluations. Though it is straightforward to simulate the VDs using software threads and processes(as described in the previous section), simulating a DA requires accurate prediction of the I/O latency of a disk access request, which is acknowledged to be a challenging task in multiple prior research works [107, 186, 199–202]. Fortunately, for RLB evaluation, an accurate estimation of the I/O latency is not a strict requirement. It is sufficient if the estimated I/O latency is a good approximate that varies according to the variations in the input workload presented to a DA. Therefore, we built a disk array simulator that calculates the approximate I/O latency of each disk access request which is represented in the form $\langle r/w, disk\ offset, size \rangle$, where r/w flag indicates whether the request is of read or write type, disk offset indicates the LBA of the I/O request in the DA and size indicates the size in kilo bytes of the I/O request.

Since a DA is just a bunch of disks that are flexible to be configured in different ways, for our experiments we chose to use a software RAID0 setup. With a Raid0 configuration, for any given I/O request, it is possible to de-

terministically ascertain the target disk drive in a DA using simple module arithmetic. When a VD submits an I/O request to a DA, that request is inserted into a FIFO queue corresponding to the particular disk drive inside the DA and the VD thread returns to submit its next request without waiting for any acknowledgement. Each DA has a dedicated thread for each disk drive in the DA and each of these threads independently dequeue from its local FIFO queue and sleeps for a time interval that is estimated to be the I/O latency experienced by that request on a hypothetical physical disk drive. In this simulation, if the requests were to be processed using a dual queue, large number of threads would have been necessary to submit multiple I/O requests in parallel, so that NCQ in the disk drive is utilized to the best possible extent. Using a single FIFO queue in the simulation helps reduce the number of threads to a great extent, but in order to match the I/O latency on a physical drive, the simulated DA uses intelligent techniques to simulate the merging and reordering operations of a NCQ on a physical drive.

The CFVC scheduler on a physical DA maintains a dual queue consisting of a CSCAN queue and a CFVC queue. In order to simultaneously ensure maximal bandwidth utilization and minimum worst case latency on a DA, the CFVC scheduler makes its best effort to process the I/O requests in sequential order of the LBAs using the CSCAN queue, unless there exists a request in the CFVC queue that exceeds its waiting time beyond a threshold time T1, because of which the CFVC queue is given higher priority and all such delayed request are processed out of order. If multiple VDs submit I/O requests simultaneously, then its likely that a group of requests from a particular VD exceed their waiting time beyond the threshold time T1. As a result the CFVC scheduler processes such requests out of order using the CFVC queue, but since these group of requests are also present in the same sequential LBA order in the CSCAN queue, the large seek latency incurred to process the first request in the group is amortized with the rest of the requests in the group. Threshold T1 is directly proportional to the worst case latency, but it is difficult to control in the simulated DA because requests are processed strictly in FIFO order. However, based on our observations on a physical DA over different set of experiments, the average size of a group of requests from a VD that are processed together on a DA is approximately around 5. Since this group size is just a heuristic based on actual observations on a physical DA, the estimated I/O latency on the simulated DA doesn't correlate accurately with the actual I/O latency, but it does correlate according to the changes in the workload on a DA.

The simulated DA calculates the I/O latency as the sum of transfer latency, rotational latency and seek latency. Transfer latency is the easiest of the three

as its independent of the workload variations. Assuming that a 7200 RPM disk is expected to support approximately 100 MBPS sequential bandwidth, the transfer latency is calculated as follows:

$$Transfer\ Latency = (IO\ request\ size\ in\ KB/102400\ KBPS) * 10^6 \mu seconds$$

Seek latency and rotational latency are inter-dependent because of the merging and reordering effects in the simulated DA. The average rotational latency on a 7200 RPM disk is expected to be 4ms, but for a merged request there is no rotational latency. For all practical purposes, we assume a maximum of 10 VDs shared by a DA and hence each VD occupies 100 GB range on a 1 TB disk. Since in the simulated DA setup, each disk has its own dedicated scheduler thread, having multiple disks in a DA doesn't stress any component of RLB but only adds to higher CPU contention. Hence the simulated DA uses just one virtual disk with a capacity of 1TB. We conducted various experiments to estimate the I/O latency for 2 cases: 1) process consecutive random I/O requests from the same VD and 2) process consecutive random I/O requests that are from different VDs. Case 1 corresponding to intra-VD latency, resulted in an average I/O latency of 4.3ms. Case 2 corresponding to inter-VD latency, was surprising because irrespective of the difference in offsets between requests from two VDs, the average I/O latency was approximately 15 ms. This is possible due to advanced mechanical techniques in modern disk drives, and hence subtracting an approximate 4.3ms rotational latency and 0.625ms transfer latency for a 64KB I/O request, from the observed 15ms average I/O latency, the average seek latency can be approximated to be 10ms.

Since the simulated DA processes requests strictly in FIFO order (as described in section 7.8.1), the order in which requests are processed on a CFVC scheduler are completely different from the order in which requests are processed from the FIFO queue in simulated DA and hence the finish time of a request has no consequence to the latency estimation in simulated DA. The pseudo code for simulated DA's latency estimation algorithm is described in Algorithm 2.

In Algorithm 2, the state variables are used to track the workload pattern from every VD on a DA, and the Min_Raw_Band_Latency is calculated as the transfer latency for the size of a block, which is 64 KB in all our experiments. The heuristics Max_Merge_Threshold_Latency, Max_Merge_Count and Max_Merge_Count_High_Rate are derived based on a trial and error basis. Using the observations of the I/O latency pattern on a physical DA with CFVC scheduler, simulated DA is fine-tuned to ensure that the estimated I/O latency doesn't deviate beyond acceptable limits from the actual I/O latency

Algorithm 2 Pseudo algorithm to estimate seek and rotational latency for an I/O request in DA simulator

INPUT:
The VD that submitted the given I/O request R: Cur_VD;
The LBA of given I/O request R: Cur_LBA;
The arrival time of given I/O request R in the DA's FIFO queue: Cur_AT;
OUTPUT:
The seek latency for given I/O request: Seek_Lat;
The rotational latency for given I/O request: Rot_Lat;
CONSTANTS:
INTER_VD_SEEK_LATENCY = 10ms;
INTRA_VD_SEEK_LATENCY = 4.3ms;
The minimum latency between requests that guarantee max utilization of raw bandwidth: Min_Raw_Band_Latency;
The maximum size of a single track: Max_LBA_Single_Track=2MB;
The maximum latency to wait to merge requests in the hypothetical CFVC scheduler: Max_Merge_Threshold_Latency=100ms;
The maximum number of requests which can be processed sequentially in a hypothetical CFVC scheduler: Max_Merge_Count=5;
The maximum number of requests which can be processed sequentially in a hypothetical CFVC scheduler when requests occur with an inter arrival latency lesser than Min_Raw_Band_Latency : Max_Merge_Count_High_Rate = 20;
STATE VARIABLES:
Merge_Count for every VD i on the DA: $Merge\_Count_i$;
The arrival time of previous I/O request processed for every VD i on the DA: $Prev\_AT_i$;
The LBA of previous I/O request processed for every VD i on the DA: $Prev\_LBA_i$;
The VD whose request was last processed on the DA: Prev_VD;

1: Assign absolute difference of Cur_LBA and $Prev\_LBA_{Cur\_VD}$ to LBA_diff;
2: Assign the difference of Cur_AT and $Prev\_AT_{Cur\_VD}$ to AT_diff;

```
3: if Cur_VD == Prev_VD then
4:     if LBA_diff <= Max_LBA_Single_Track then
5:         if AT_diff <= Min_Raw_Band_Latency then
6:             Assign 0 to Rot_Lat;
7:             Increment Merge_Count_Cur_VD by 1;
8:         else
9:             if AT_diff > Max_Merge_Threshold_Latency then
10:                Assign 0 to Merge_Count_Cur_VD;
11:                Assign FULL_ROT_LATENCY to Rot_Lat;
12:            else
13:                if Merge_Count_Cur_VD < Max_Merge_Count then
14:                    Increment Merge_Count_Cur_VD by 1;
15:                    Assign 0 to Rot_Lat;
16:                else
17:                    Assign 0 to Merge_Count_Cur_VD;
18:                    Assign FULL_ROT_LATENCY to Rot_Lat;
19:                end if
20:            end if
21:        end if
22:        Assign 0 to Seek_Lat;
23:    else
24:        Assign 0 to Merge_Count_Cur_VD;
25:        Assign INTRA_VD_SEEK_LATENCY to Seek_Lat;
26:        Assign FULL_ROT_LATENCY / 2 to Rot_Lat;
27:    end if
28: end if
```

```
29: if Cur_VD ! = Prev_VD then
30:     if LBA_diff <= Max_LBA_Single_Track then
31:         if AT_diff > Max_Merge_Threshold_Latency then
32:             Assign 0 to Merge_Level;
33:         else
34:             if Merge_Count_Cur_VD < Max_Merge_Count then
35:                 Assign 2 to Merge_Level;
36:             else
37:                 if AT_diff <= Min_Raw_Band_Latency and
    Merge_Count_Cur_VD < Max_Merge_Count_High_Rate then
38:                     Assign 2 to Merge_Level;
39:                 else
40:                     Assign 0 to Merge_Level;
41:                 end if
42:             end if
43:         end if
44:     else
45:         if AT_diff <= Max_Merge_Threshold_Latency then
46:             Assign 1 to Merge_Level;
47:         else
48:             Assign 0 to Merge_Level;
49:         end if
50:     end if
51:     if Merge_Level == 2 then
52:         Increment Merge_Count_Cur_VD by 1;
53:         Assign 0 to Seek_Lat;
54:         Assign 0 to Rot_Lat;
55:     else if Merge_level == 1 then
56:         Increment Merge_Count_Cur_VD by 2;
57:         Assign INTRA_VD_SEEK_LATENCY to Seek_Lat;
58:         Assign FULL_ROT_LATENCY / 2 to Rot_Lat;
59:     else
60:         Assign 0 to Merge_Count_Cur_VD;
61:         Assign INTER_VD_SEEK_LATENCY to Seek_Lat;
62:         Assign FULL_ROT_LATENCY / 2 to Rot_Lat;
63:     end if
64: end if
```

as observed on a physical DA. The following subsection demonstrates the correctness of this simulation algorithm by comparing it to the real DA.

Performance Evaluation of Simulated DA

Real DA		**Simulated DA**	
RGN	OT	RGN	OT
50.54	50.49	49.37	48.16
1	1	1	0.98
1	1	1	0.98
1	1	1	0.98

Table 7.1: *Table showing the correctness of simulated DA by comparing the performance to real DA. Request generation rate (RGN) and Observed Throughput (OT) are measured in units of MBPS*

Real DA		**Simulated DA**	
RGN	OT	RGN	OT
44.44	42.33	41.59	38.4
10	9.1	9.82	8.31
10	9.1	9.82	8.31
10	9.0	9.82	8.31

Table 7.2: *Table showing the correctness of Simulated DA by comparing to Real DA, using negative result. Request generation rate (RGN) and Observed Throughput (OT) are measured in units of MBPS*

Table 7.1 shows 4 VDs configured with 70% sequential locality, 70% read I/O requests, three VDs generate requests at a rate of 1 MBPS and one VD generates requests at a rate of 50 MBPS. While the real DA shows that the observed throughput (OT) lags the request generation rate (RGN) by less than 1%, the simulated DA lags by less than 2.5%. Such a close correlation between the results from real DA and simulated DA suggests the effectiveness of the simulation algorithm. However, to prove the correctness of the simulation algorithm, we overloaded the DA as shown in Table 7.2. It can be seen that in both the real DA and the simulated DA, the RGN for the first VD is much lower than the actual generation rate of 50 MBPS because the four VDs collectively overload both the real DA and simulated DA. Though, the drop in OT when compared to RGN is around 8% on real DA, and around 15%

on the simulated DA, the fact that the simulated DA successfully recreated the overloaded situation proves the correctness of the simulation algorithm. Therefore, as expected the above two experiments show that though the simulation algorithm doesn't accurately predict the I/O latency, it is well within acceptable error rate, and hence is good enough to measure the effectiveness of RLB algorithm.

7.8.3 Synthetic Trace Generation

Finding a publicly available real-world trace workload from standard applications like a web-server or an email server is extremely difficult. In order to effectively evaluate the RLB algorithm, we need a massive setup with several VDs and each VD should generate unique workload. Therefore, we developed a synthetic test suite that can generate workloads with the following variables: percentage of read I/O requests, request generation rate in units of KBPS, request size in units of KB, locality expressed as percentage of sequential requests, information of the target DA for the given VD in terms of networking port number and starting offset for that VD on that DA. Each VD has a dedicated thread that generates workload according to a preconfigured setting using the variables mentioned above. VDs that share a DA are mapped to unique portions of the disk and hence each VD needs to know the starting offset for each of its replicated DAs. The request generation rate can either follow a constant frequency distribution or can follow a Poisson distribution. The percentage of sequentially located requests directly determines the amount of randomness in the generated workload. It is possible to generate requests one by one, in strict adherence to the randomness factor, or in clusters of group size ranging randomly between one to eight, which is most typical in a real-world scenario. We choose the latter, and only the LBA of the first request in the group is assigned either sequentially with respect to the previously generated group, or randomly to a location on the DA that is within the LBA range that the given VD is authorized to store data on that DA.

In order to simulate a real-world trace, the read/write ratio, sequential locality and the request generation rate should vary independent to each VD, but should remain consistent within a VD. Therefore we pre-configure every VD with the following default settings unless they are explicitly mentioned: request size of 64 KB, request generation frequency follows Poisson distribution, loopback ethernet address is used for target DA, sequential locality of 70%, read/write ratio of 70% and the request generation rate is varied for each VD. Since the sequential locality and read/write ratio is configured for 70% and each VD follows a different random distribution, the workload patterns

in each VD are unrelated, but within a VD, the workload pattern varies in an expected manner and therefore if the sampling duration is long enough, *Cheetah* effectively captures the overall picture of the workload.

7.9 Performance Evaluation of the Automated PB Extraction Process

There are two important issues in evaluating the PB extraction process. First, it is necessary to demonstrate that the PB value indeed captures the overall characteristics of a VD's I/O workload. Second, it is important to prove the correctness of the PB extraction process. To handle the first issue, we use standard real-world traces and synthetic workloads to show the variations in PB value in different kinds of I/O workloads. The second issue is more complicated because it is difficult to prove the correctness of the PB extraction process, as it is not clear as to what defines a correct PB value. Due to advanced optimizations on modern disk drives, a DA can satisfactorily handles multiple VD workloads, whose sum of PB values is greater than the raw bandwidth capacity of the DA. The CFVC scheduler too contributes to these optimizations by converting random I/O requests into mostly sequential I/O requests, whenever possible. One possible way to demonstrate the correctness of the PB extraction process is to show its effectiveness in a DA that processes multiple VD workloads with different workload localities. Therefore, in this evaluation study, rather than addressing the correctness of the PB extraction process, we focus on proving its effectiveness.

7.9.1 Effect of Workload Locality on PB using Real-World I/O Trace

We used the real-world traces from Umass [203] that provide five different I/O traces, of which two traces are collected from online transaction processing (OLTP) applications running at two large financial institutions and three traces are collected from a popular web search engine. The OLTP traces provide a good mix of reads and writes, while the web search traces are read heavy. The data locality in these traces are evenly spread out in the range of 0 to 2 TB. From the trace data, for each entry corresponding to an I/O request, we only considered the LBA, read or write type, size of the I/O request and the relative timestamp at which the I/O request appeared in the workload. We used the prototype described in Section 7.8.1 to evaluate each of these traces one at a time and extracted their PB value using a real physical DA. *Cheetah*

Workload	RGN	PB	R/W
OLTP1	3.15	63	1.5
OLTP2	8.04	148	5.9
WebSearch1	13.46	96	909
WebSearch2	13.56	83	833
WebSearch3	8.09	69	714

Table 7.3: *Table showing the effect of workload locality on PB value using 5 different real-world traces. RGN and PB are measured in units of MBPS. Read / Write ratio (R/W) is expressed as the ratio of number of reads to number of writes in the trace.*

computes the PB value on the DA, after observing the data locality pattern in the incoming I/O workload for a default sampling period of one minute. The DA then uses the CFVC scheduler to process the I/O requests based on the extracted PB value. Table 7.3 shows the variations in PB value for each of these five real-world traces, when considered independently. *Cheetah* measures the request generation rate (RGN) and $\frac{Read}{Write}$ ratio within the VD, and measures the PB on the DA.

Workload	RGN	PB	R/W
OLTP1	8.08	101	1.5
OLTP2	16.30	204	5.9
WebSearch1	23.9	109	909
WebSearch2	21.5	107	833
WebSearch3	15.8	94	714

Table 7.4: *Table showing the effect of workload locality on PB value using 5 different real-world traces, when RGN is doubled. RGN and PB are measured in units of MBPS. Read / Write ratio (R/W) is expressed as the ratio of number of reads to number of writes in the trace.*

The relative timestamp field in the trace data depicts the request arrival distribution pattern and without loss of generality it is possible to uniformly vary this RGN value to study the corresponding variations in PB value. Along these lines, we halved the difference between successive I/O request's timestamps in the hope of increasing RGN by a factor of two and Table 7.4 shows the corresponding PB variations. It can be seen that RGN does increase by a factor of two, but even though PB value increases, it doesn't increase in proportion to the increase in RGN value. The rate of change in PB value is higher in OLTP trace, than in the web search trace because the web search trace is predominantly occupied with read I/O requests and naturally has higher

chances of maximizing the caching opportunity in both the OS buffer cache and the disk's cache. However, the OLTP trace has a good number of write I/O requests, which unfortunately cannot exploit any caching benefits because all the caches on the write path are disabled for data consistency purposes. In addition to these differences, the PB value also varies with the amount of randomness in the trace data's locality. It is quite possible for the web search traces to have a large number of consecutive sequential I/O requests and higher timestamp differences between them. Due to this, even though the consecutive requests are sequential, they incur full rotational latency because the disk drives cannot wait in anticipation for merging and reordering opportunity beyond a threshold time. When the RGN value is doubled, many such requests could possibly be processed sequentially, leading to lesser disk resource utilization time for the web search traces. Hence for these web search traces, PB value increases sub-linearly when RGN is doubled.

7.9.2 Effect of Workload Locality on PB using Synthetic Workload

With real-world traces, we observed few interesting patterns in workload locality that reflected in variations in PB value, but due to limited configurable parameters in the real-world trace, it was not possible to comprehensively evaluate the PB extraction process. Therefore, in this section, we use synthetic workloads to vary RGN, sequential locality and read/write ratio to get a detailed understanding of the nature of variations in PB value in different types of workload.

Effect of RGN on PB

RGN	390.31	314.34	204.15	101.28	50.16	9.98	1
PB	426	359	294	361	313	90	10

Table 7.5: *Table showing the variations in PB with variations in RGN. All units are in MBPS*

Table 7.5 shows the variations in PB when requests are generated at different rates. All workloads are configured with 100% sequential locality and 100% write requests. The DA consists of four disks in RAID 0 configuration. When a VD is configured to generate requests at a very high rate of 400 MBPS, though the DA is capable of handling the load, the loopback network interface throttles the generated rate occasionally and hence we see the RGN

value of 390.31. This is inconsequential because the drop in throughput is only marginal and doesn't affect the results of the performance evaluation. The table shows that as RGN drops, PB drops too, but the rate at which PB drops is much slower than that of the RGN. The reason is because when requests are generated at a slower rate, the inter-arrival I/O latency between successive requests increases and hence the chances of merging and re-ordering I/O requests on either the CFVC scheduler or on the NCQ of the disk drives in the DA diminishes. It is also a likely possibility that when RGN decreases, the chances of full rotational latency on the disk drives increase, similar to the observations made on the real-world trace in the previous subsection. Therefore, though the PB value decreases, it decreases at a slower rate compared to that of RGN.

Effect of Sequential Locality on PB

Locality	100%	70%	10%
PB	313	342	374

Table 7.6: *Table showing the variations in PB when workload locality is varied. Locality is expressed as percentage of sequential requests in the workload and PB is in units of MBPS*

Table 7.6 shows the variations in PB value when the locality expressed as the percentage of sequential requests in the workload, is varied between 100% to 10%. RGN is fixed to 50 MBPS and 100% write requests throughout this experiment. As expected, when the percentage of randomness in the input workload increases, the average disk I/O latency increases and hence the PB value increases. However, the rate at which PB increases is much slower than the rate at which the sequential locality in the workload decreases. There are a couple of reasons for such a behavior. First, the RGN value of 50 MBPS is much lower than the full raw disk bandwidth and hence majority of the requests experience full rotational latency even with 100% sequential locality. Second, the CFVC scheduler has the ability to process the I/O requests as sequentially as possible and hence amortizes the randomness factor to some extent.

Effect of Read/Write Ratio on PB

Table 7.7 shows the variations in PB value when the read/write ratio expressed as the percentage of reads in the overall workload, is varied between 100% to

R/W ratio	100%	50%	10%	0%
PB	381	347	336	313

Table 7.7: *Table showing the variations in PB when read/write ratio is varied. Read/Write ratio is expressed as the percentage of reads in the overall trace. PB is in units of MBPS*

0%. RGN is fixed to 50 MBPS and locality to 100% sequential throughout this experiment. Surprisingly, when the percentage of read I/O requests decreases in the workload, PB value decreases. This suggests that the read cache did more harm than good and upon a deeper look into the experiment settings, it becomes clear that the read cache and prefetching optimizations done by the DA isn't useful for a sequentially generated workload that doesn't reuse any data from cache. Though a real-world workload benefits from read cache, this particular synthetic workload doesn't benefit much and as a result, the DA does redundant work in fetching additional data from disk for no benefit, resulting in an increase in PB value. To cross check, we disabled all the caches in the entire data path and repeated these experiments to find that the PB value doesn't vary beyond 1%. Therefore, this experiment suggests that read/write ratio in the I/O workload has negligible effect on the PB value if the read requests do not make use of the cache.

7.9.3 Effectiveness of PB Extraction Process on a Shared DA

RGN	50.54	1	1	1
OT	50.49	1	1	1
PB	359	160	160	160

Table 7.8: *Table showing the effectiveness of PB extraction process on a DA shared by four VDs with different workload localities. OT, RGN and PB are all measured in units of MBPS*

The effectiveness of PB extraction process can be effectively demonstrated when a DA is shared my multiple VDs with different workload localities. Additionally, the CFVC scheduler calculates the finish time of an I/O request based on the PB value of a VD, and hence an incorrect PB value will very likely result in an VD receiving less than its expected bandwidth. Therefore, the effectiveness of extracting PB from an application's sample workload is

demonstrated when multiple VDs with different workload patterns simultaneously share a DA and yet each VD receives the expected bandwidth. Table 7.8 shows four VDs configured with the following settings: 70% sequential locality, 70% read I/O requests. The statistics collected from CFVC scheduler for each of the VDs are shown in each row separately. The experiment shows that even when one of the VDs generates requests at a rate of 50 MBPS and the other three VDs generate requests at a rate of 1 MBPS each, the OT values as measured by each of the VDs individually, has less than 0.1% variations from their corresponding RGN values. Such a close correlation between the RGN and OT values suggests, both the accuracy in PB value extraction and the implementation correctness of the CFVC scheduler.

RGN	44.44	10	10	10
OT	42.33	9.1	9.1	9
PB	356	205	206	206

Table 7.9: *Table showing the effectiveness of PB extraction process on a overloaded DA shared by four VDs with different workload localities. OT, RGN and PB are all measured in units of MBPS.*

Table 7.9 shows the result of overloading a DA with multiple workloads that collectively generate more load that the DA can handle, and hence the OT differs significantly from its RGN. The experiment setup is similar to the previous experiment explained in table 7.8, except that the three VDs generating at a rate of 1 MBPS each are configured to generate the workload at a rate of 10 MBPS each. Since the DA is 100% busy and is clearly overloaded, the VD generating at a higher rate of 50 MBPS employs a naive congestion control algorithm to block itself until the number of pending I/O requests are not too high (we have heuristically chosen 10K as the threshold limit). As a result the RGN value of that VD is just 44.44 MBPS and the OT is even lesser at 42.33 MBPS. Though the PB value for each of the four VDs are accurately determined, incorrect admission control policy and the absence of flow control policy led to an overloaded DA and eventually the bandwidth guarantee cannot not be satisfied accurately. This experiment demonstrates that even in an overloaded DA, due to effective PB extraction process, *Cheetah* allocates proportional disk bandwidth to each of the VDs that share the overloaded DA.

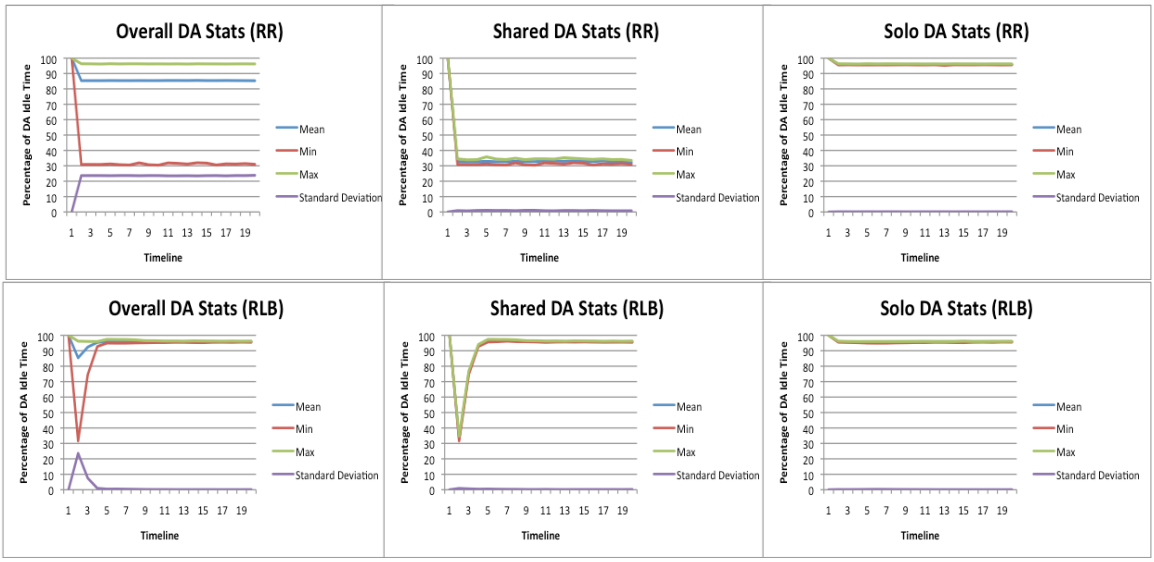

Figure 7.8: Charts with load distribution for uneven VD-DA mappings

7.10 Evaluation of the Effectiveness of Bandwidth Decomposition

In the following experiments, we demonstrate the RLB's effectiveness by varying the workload locality and mapping between VDs and DAs in various ways. All the figures below are represented with the following semantics. The Y-axis shows the amount of idle time when the DA does no useful work and its expressed as a percentage of the total observation time. Since the number of DAs are too high, we highlight only the the DAs with minimum and maximum idle time percentage. We also show the mean and standard deviation of the idle time percentage over all the DAs, to better understand the effect of RLB on all the DAs. On the Y-axis, a low value indicates an over-loaded DA and a high value indicates an under-loaded DA. The X-axis represents the time in terms of epochs, where each observation point corresponds to a single window of time in which RLB collects DAs' loads and capacities and calculates a distribution weight for all the VDs. Statistics were collected from every DA for every 1 minute over a 20 minute period.

7.10.1 Variable VD-DA Mappings

Consider a setup, where for each VD, one of the replica DAs is dedicated only to this VD and the other two replica DAs are shared by nine other VDs. On

such a setup, the RLB algorithm is stressed to a great extent because of the obvious opportunity to load balance the DAs. To evaluate the effectiveness of the RLB algorithm, we compare it with a round robin (RR) scheduler. The RR scheduler always distributes the read load on each VD into equal portions among all the replica DAs. All VDs were configured with the same workload consisting of: 100 VDs, 120 DAs, 100% read request type, 100% sequential locality, 64KB request size, 1280 requests/min. 100 DAs were marked as Solo DAs and the remaining 20 Shared DAs were mapped with 10 VDs each. Figure 7.8 shows 6 graphs to give a detailed analysis of the RLB's effectiveness. The graphs are primarily characterized as 3 graphs for RR on the top and 3 graphs for RLB on the bottom. The set of dedicated DAs that are mapped with only a single VD are marked as Solo, and the remaining DAs that are shared by 10 VDs are marked as Shared.

For the first two observation points until the RLB evaluates the distribution pattern, local scheduler on each VD uses RR scheduling by default. Later on, when the RLB collects the actual load on the DAs and the VDs use this information to load balance their workload, we can see the differences in performance between RLB and RR schedulers. When all the DAs are considered together, we see that for the RLB scheduler, the min and max curves are very close to each other and hence the standard deviation of the idle time percentage among all the DAs is almost 0. However, with a RR scheduler, the mean, min and max curves are wide apart and hence the standard deviation is noticeably high. When only the Solo DAs are considered for analysis, there isn't much to distinguish between the RLB and RR schedulers, because for the Solo DAs there isn't much variations in the workload. When only the Shared DAs are considered for analysis, the RR scheduler overloads all such Shared DAs and as a result the mean, min and max curves show that they are all over-loaded, when compared to that of the RLB scheduler. Since the RLB scheduler redirects the load from potentially overloaded Shared DAs to the Solo DAs, none of the Shared DAs are overloaded in the RLB scheduler.

In the previous experiment, all the VDs were configured with identical workloads and more importantly all the shared DAs were mapped with the same set of VDs, which means given two shared DAs, they accommodated the same set of ten VDs. Such a monotonous mapping was the reason why even a RR scheduler could ensure evenly loaded shared DAs. In this experiment, we stress the RLB scheduler by varying the request generation rate on each VD and we further randomized the VD-DA mappings so that the load on each DA is uneven and unpredictable. The request generation rate is chosen among the set of 5 values starting from 256KBPS in intervals of 256 KBPS. The other workload parameters were configured the same for all VDs, and consisted

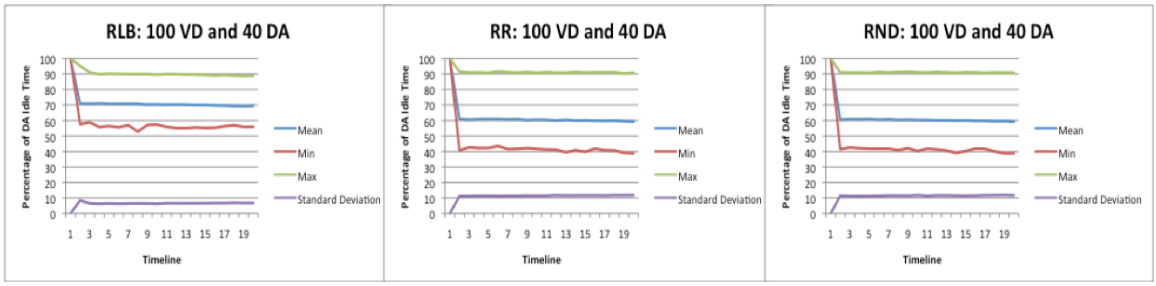

Figure 7.9: Figure comparing RLB, RR and RND schedulers for random mappings between 100VDs and 40 DAs

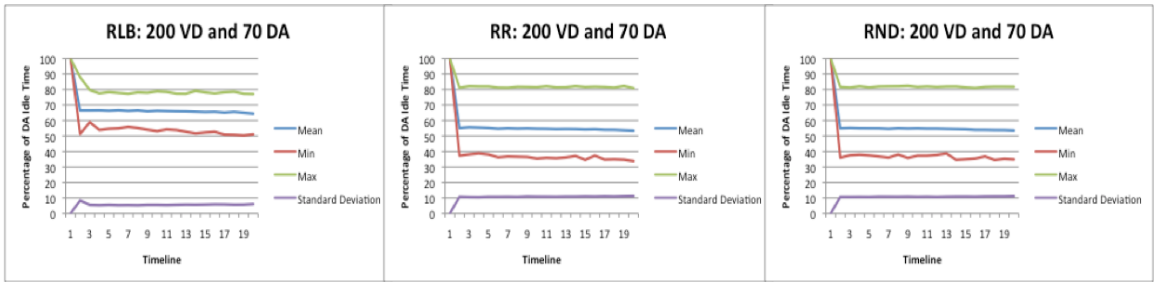

Figure 7.10: Figure comparing RLB, RR and RND schedulers for random mappings between 200VDs and 70 DAs

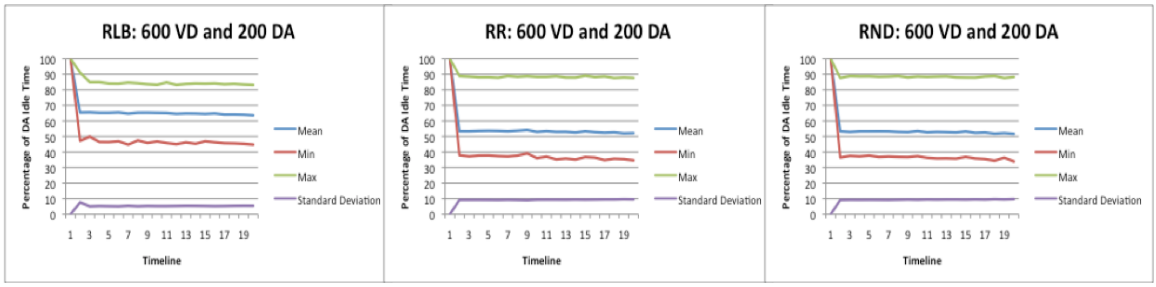

Figure 7.11: Figure comparing RLB, RR and RND schedulers for random mappings between 600VDs and 200 DAs

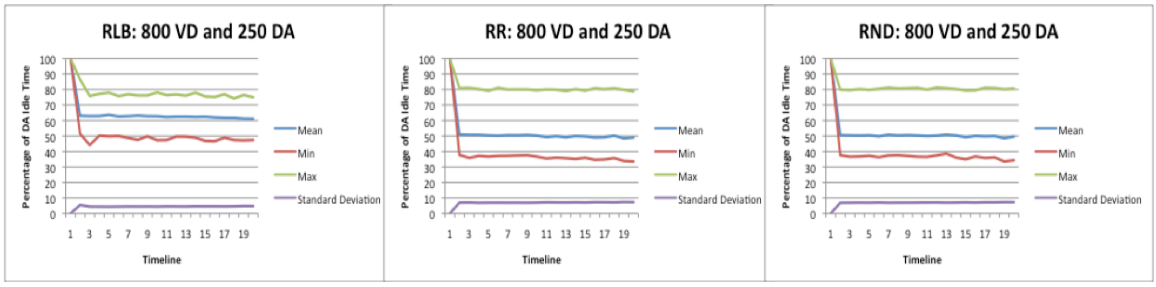

Figure 7.12: Figure comparing RLB, RR and RND schedulers for random mappings between 800VDs and 250 DAs

of 70% sequential locality, 70% read I/O requests, 64KB request size, Poisson frequency distribution for request generation. We varied the VD-DA mappings for 4 different values: 100VD-40DA, 200VD-70DA, 600VD-200DA and 800VD-250DA. The mappings between VDs and DAs are completely randomized, except for the fact that a VD is mapped to 3 different replica DAs. We further compare RLB scheduler with not just the RR scheduler but also with a RND scheduler that selects the replica DAs in a random order.

Figures 7.9 - 7.12 show 3 graphs each for RLB, RR and RND schedulers. Both the RR and RND schedulers show identical results and that can be attributed to two reasons: 1) Simulated DA aggregates the requests from different VDs and processes them mostly sequentially. 2) RLB collects load information from all the DAs not too frequently. For these two reasons, any short term fluctuations in load distribution will have negligible impact on the DA's performance and hence even lesser impact on the RLB load distribution efficiency. When compared to RR or RND scheduler, the RLB scheduler delivers an impressive performance with standard deviation lowered by approximately 44% and hence the mean, min and max curves for RLB scheduler are much closer when compared to that of RR or RND scheduler.

7.10.2 Variations in Read/Write Ratio

The RLB scheduler doesn't make any intelligent decision for write I/O requests, but the presence of write I/O requests changes the load on the DAs by a substantial margin because every write I/O request generated by a VD results in three write I/O requests on three different DAs. In this experiment, we use 31 DAs and 100 VDs and keep the rest of the setup similar to the previous experiment, except for the variations in read/write ratio. We vary the read I/O percentage between 0 to 100 to study their effects on RLB scheduler. Figure 7.13 shows 6 graphs with different read percentage variations. Upon decreasing the percentage of reads from 100% to 0%, we see a gradual increase in standard deviation, because a decrease in read I/O requests, decreases the chances for RLB to balance the load distribution. Hence the DAs tend towards unbalanced condition with the increase in write percentage.

7.10.3 Variations in Sequential Locality

RLB scheduler is expected to balance the DA loads without any dependency on the locality in a VD's workload. The experiment setup is similar to that in the previous experiment, except that the read/write ratio is fixed at 70% and the sequential locality is varied between 0 to 100. Figure 7.14 shows the load distribution pattern for 6 different variations in sequential locality in the input

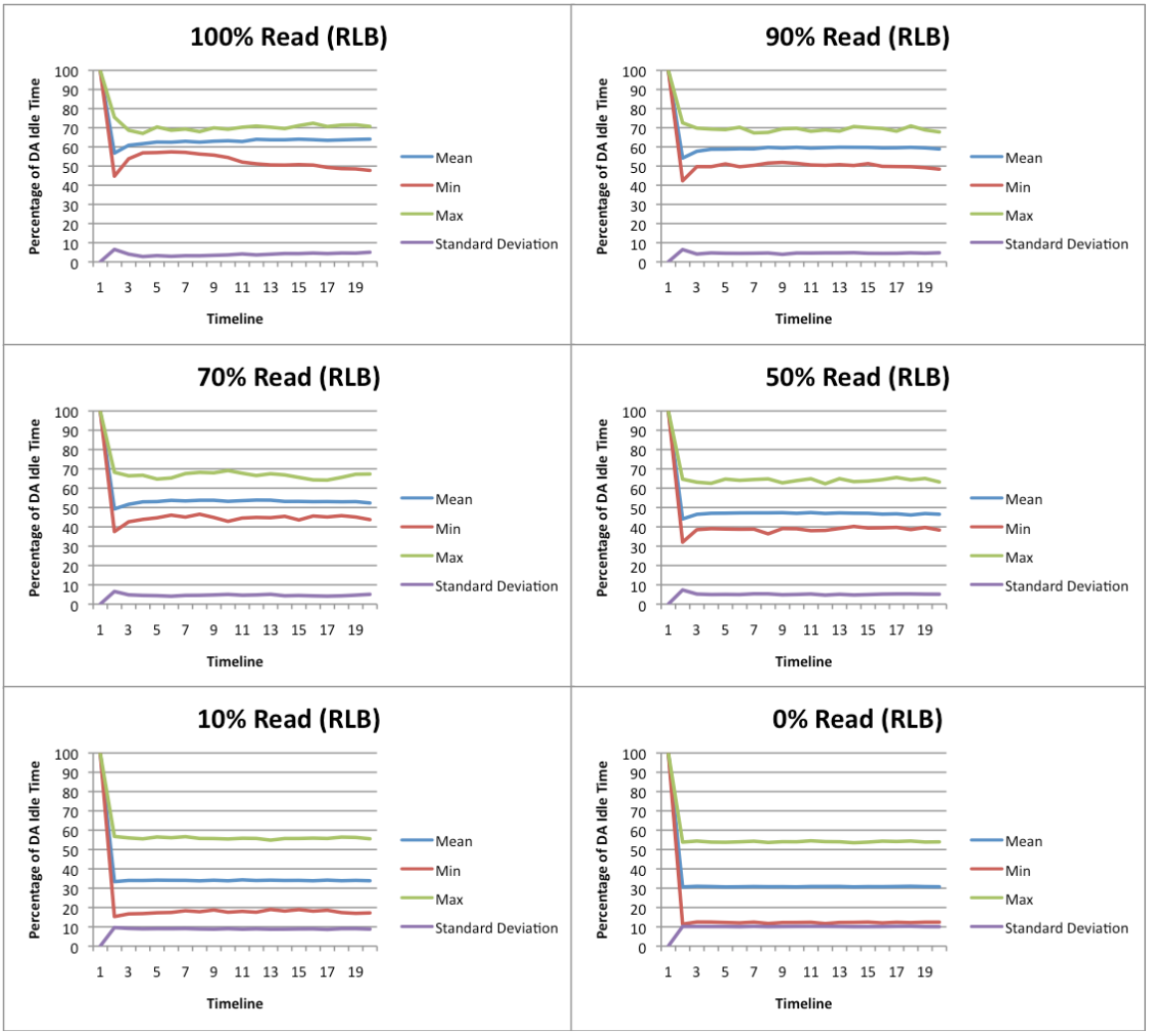

Figure 7.13: *Charts with load distribution for variations in read/write ratio. Comparison between different read percentage variations and between RLB, RR and RND schedulers*

workload. Across all variations, the standard deviation is consistently between 4.16 to 7.75 and this low standard deviation suggests that the RLB algorithm has very minimal interference from the locality in the input workload. One of the primary reasons for the minor deviations in standard deviation is due to the lack of a perfect DA simulator. However, the consistent expected results across all the variations in sequential locality suggests that the DA simulation indeed delivered a good approximation to the CFVC scheduler. The figure also shows that as the sequential locality decreases from 100% to 0%, the mean of

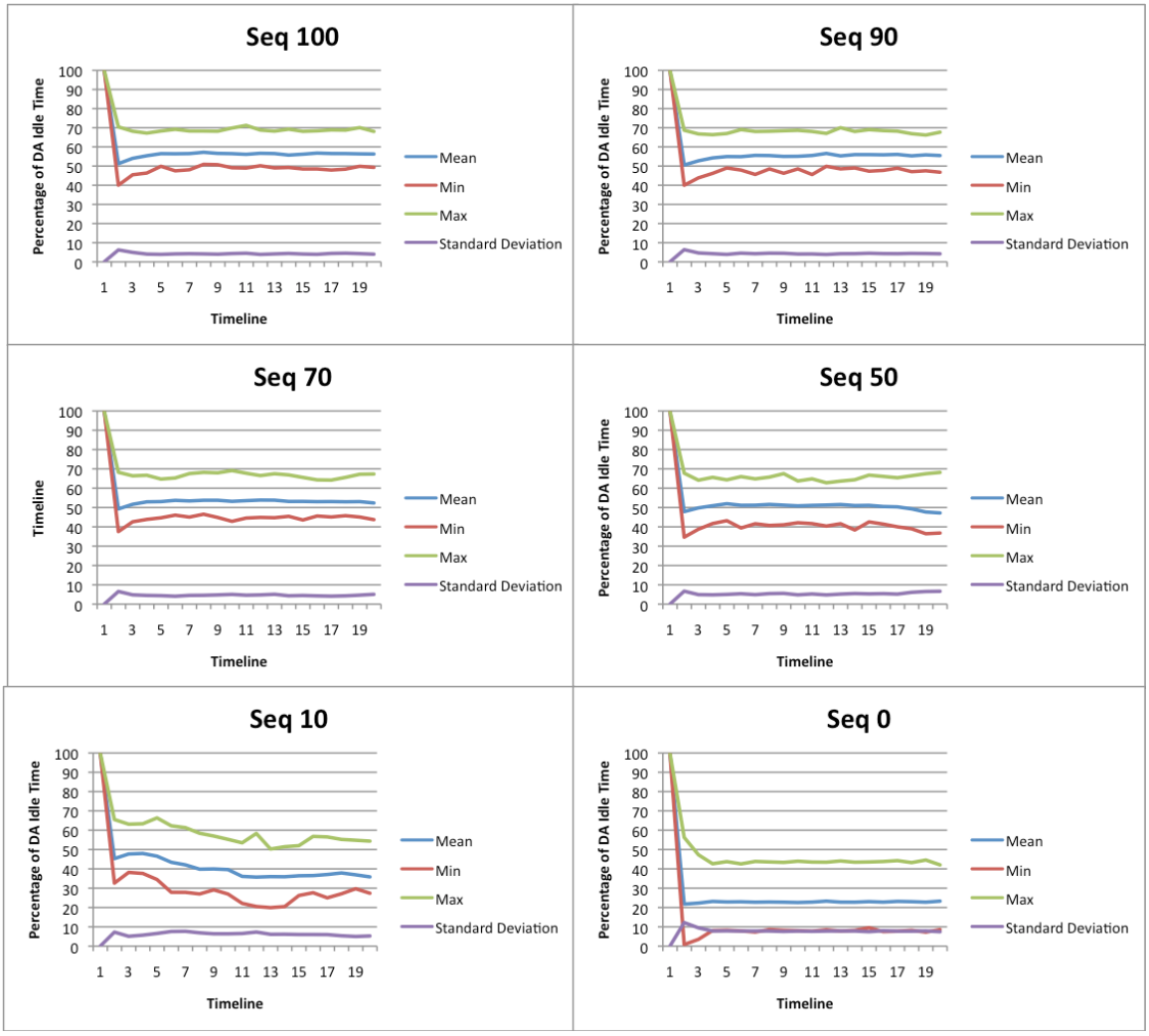

Figure 7.14: Charts with load distribution for variations in sequential locality in input workload

the disk idle time percentages among all the DAs also decreases. However, it is interesting to note that even though the random content in the workload increases by 50% between 100% and 50% sequential locality, the mean curve doesn't drop down drastically. The increased random locality is expected to increase the average I/O latency of an I/O request substantially, but because each I/O request occurs in group of sizes between 1-8, where the group size is determined by a random variable, the random seeks of the head request in a group are amortized with the other requests in the group. Further, the

DA simulator mimicking the CFVC scheduler, processes the I/O requests in mostly sequential order and hence unless the degree of randomness is too large, the DA idle time percentage doesn't rapidly drop down with increase in randomness in the input workload. The fact that the observed result is as expected, suggests that the simulated DA does a very good approximation to the hypothetical CFVC scheduler.

7.10.4 Short Term Variations in Workload Locality

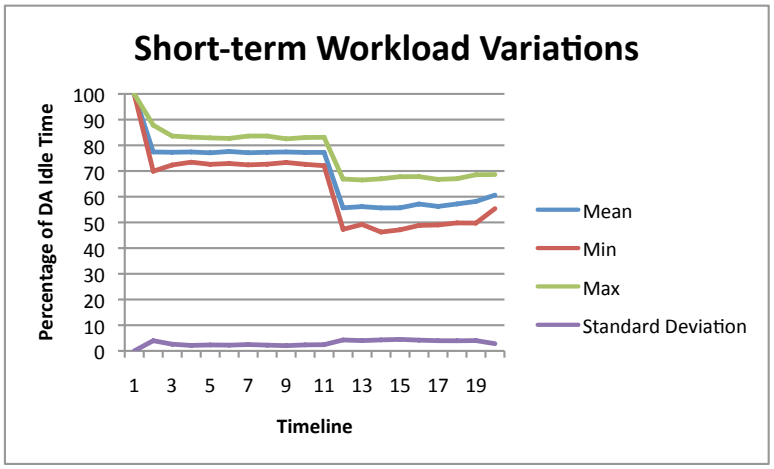

Figure 7.15: Charts with load distribution for short term variations in input workload

In all the previous experiments, though we varied various attributes in the input workload, the RLB algorithm was not tested with short term fluctuations in the input workload. It is quite common to see such short term fluctuations in a real-world workload. In this experiment, we keep read/write ratio to 70%, sequential locality to 70%, and randomly assign RGN values to VDs between 256 KBPS to 1280 KBPS, as described in the previous experiments. After 10 minutes of generating the workload, during the mid-way of our evaluation time window, we increase the RGN by a factor of 2 on all the VDs. Figure 7.15 show that at around mid-way, during the 11th minute of the evaluation window, the DAs are more overloaded due to the increased request generation rate and RLB is able to quickly adapt to the modified workload pattern. It should be noted that though each VD uses an independent random distribution to vary its attributes, RLB is still able to continuously adapt to the changes in workload pattern and ensure uniform load balancing on all the DAs. The standard deviation increases marginally due to short term fluctuation because unlike

VD Count	DA Count	RLB Time(ms)
100	120	2.5
100	40	2.5
200	70	7.5
600	200	218
800	250	455

Table 7.10: *Table showing the time taken by centralized RLB scheduler for varying number of VD-DA configurations. Time is measured in units of milli seconds*

the real DA, the simulated DA doesn't use elasticity metric to control such short term fluctuations.

7.10.5 Centralized RLB Scheduler's Processing Time

Since the RLB scheduler uses a centralized approach, it is obvious that the RLB scheduler's computation time increases with the increase in number of DAs and VDs, but it is important to ensure that the computation time doesn't increase beyond a threshold limit, for a practically large-size cloud storage system. Table 7.10 shows the time taken by centralized RLB scheduler for various VD-DA mapping combinations. We used a similar setup as described in experiment 7.10.1. Even with 800 VDs and 250 DAs in the system, the time taken by centralized RLB scheduler is 455 milli seconds, which is less than 1% of the sampling time of 1 minute, for which the statistics are collected on each DA in any given epoch. Therefore, this experiment convincingly demonstrates that the piecemeal iterative procedure adopted by the centralized RLB scheduler doesn't affect the swiftness with which the RLB scheduler recomputes the load distribution pattern in the event of load fluctuations in VDs' workloads.

7.11 Performance Evaluation of Per-VD Scheduler

Cheetah is able to enforce accurate QoS guarantees while ensuring maximum disk bandwidth utilization because the local scheduler on VD preserves the locality in the input workload and also uses the hints from the centralized RLB scheduler to ensure uniform load balancing across all the DAs. Figure 7.16 shows that a locality unaware scheduler that blindly follows the load distribution pattern as suggested by the RLB scheduler, performs relatively poorer to a local scheduler that uses both locality in the workload and the load distri-

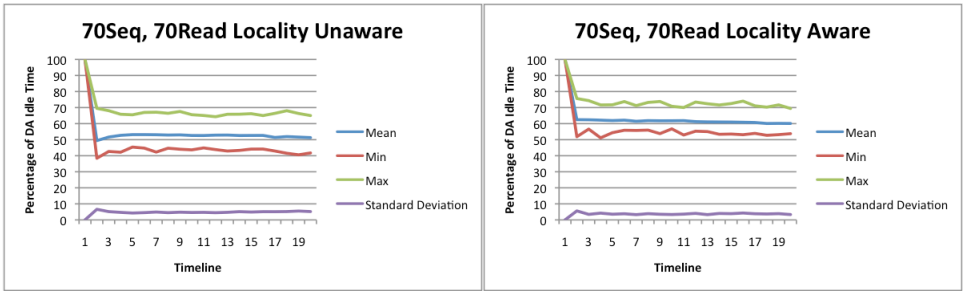

Figure 7.16: Charts comparing locality unaware vs locality aware RLB for a low locality workload consisting of 70% sequential locality and 70% read/write ratio

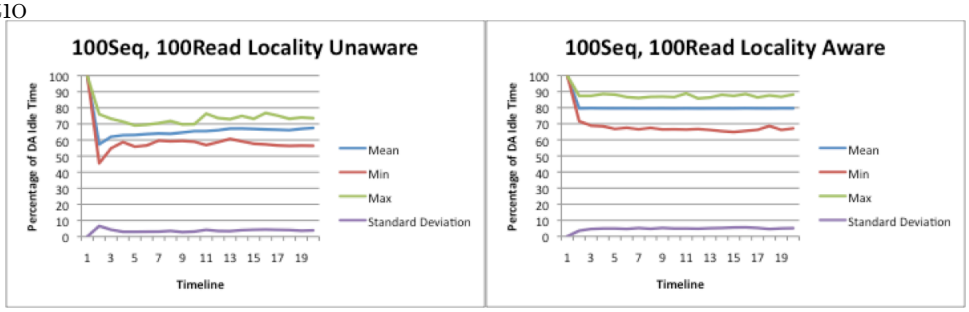

Figure 7.17: Charts comparing locality unaware vs locality aware RLB for a high locality workload consisting of 100% sequential locality and 100% read/write ratio.

bution pattern suggested by the RLB scheduler. The experiment is configured with 70% read/write ratio, 70% sequential locality and the rest of the setup is similar to the experiment described in section 7.10.3. The mean curve indicates that the average idle time of a DA increases from 52.5% in a locality unaware scheduler to 61.2% in a locality aware scheduler. The **16.6**% improvement in the idle time percentage shows the effectiveness of the locality aware scheduler. The improvement could be even better, if more intelligence is used to identify locality in the input workload. The present version is configured to maintain a LRU cache of 128 entries and every entry maintains the read I/O requests's offset in its DA along with the target DA from which it is was last referenced. When the scheduler receives a read I/O request, it first looks into this cache to pickup the best candidate DA, such that the candidate's offset is the closest to the given request and is within a threshold distance of T sectors. T is chosen to be 100 in this experiment, using the heuristic that the chosen DA could use its cache, NCQ and other advanced techniques, to process this request faster than if it were to be processed by any other replica DA. Once the local

scheduler selects a candidate DA, it submits the given read I/O request to that DA, as long as it doesn't violate the RLB's guided load distribution ratio over a larger time interval. If it does violate, then the local scheduler picks up a DA as per the RLB's load distribution ratio. Since the local scheduler is able to maintain the load distribution over a time interval that is neither too large to induce load imbalance on the DAs, nor is too small to miss the locality in the input workload, it is able to extract the maximum performance from the storage system. We also measured the end-to-end average I/O latency on each VD, and the locality aware version measured an impressive 11.866 ms against 14.956 ms in the locality unaware version. The reported latency is somewhat artificial because it doesn't take into account the network latency, which would have been present in a real system. However, the network latency is common to both the compared versions and more importantly the absolute latency is not the selling point from this experiment. It is the **26**% relative difference in I/O latency between these two versions that makes the locality aware version the clear winner.

When the locality in the input workload improves, the locality aware version shows even better improvement in the overall performance compared to a locality unaware version. Figure 7.17 shows the result when the VDs were configured with 100% sequential locality, 100% read/write ratio and the rest of the settings were kept similar to the previous experiment. Its interesting to note that though the locality aware version has a higher standard deviation of 5.0, when compared to 3.6 on the locality unaware version, the average DA idle time for locality aware version is 79.6%, when compared to 65.5% on the locality unaware version. The end-to-end average I/O latency on the VDs in the locality aware version measured an impressive 8.12 ms against 14.29 ms in the locality unaware version. Thus, on a worklod with high locality, the locality aware version performed much better than the locality unaware version with **21.5**% improvement in the DA idle time percentage and **76**% drop in the end-to-end I/O latency.

Locality-aware scheduler may exploit locality at the expense of load balance, and therefore may cause more load imbalance than the locality unaware version, even though the total load is reduced, as evidenced by lower latency, beause of locality exploitation. This experiment convincingly demonstrates that the absolute load balance is not necessarily the most important goal; being able to cut down the total load at the cost of some increase in load imbalance may be even more worthwhile.

7.12 Summary

A cloud-scale distributed storage system shared by multiple tenants require guaranteed storage performance in order to ensure fair allocation of storage resources and performance isolation to each tenant. It is an usual practice to use QoS specifications to control the performance guarantees of the storage system. However, on a massively distributed storage system there are multiple challenges that not only make it difficult for the tenants to accurately configure QoS specifications but also make it difficult for the SDDS system to simultaneously enforce QoS guarantees and ensure maximum storage hardware utilization. We propose a QoS model called *Cheetah*, that uses the following novel techniques to solve the above mentioned challenges:

- Automatically convert application-level QoS specification to physical-level QoS specification, by extracting all the necessary attributes from a small sample of the input I/O workload without any manual intervention,
- Provide QoS guarantees at both VD and VDC level of granularity.
- Dynamically decompose a VD's/VDC's QoS specification into a set of QoS specifications for each VD-DA pair, such that the locality in the input workload is captured and none of the corresponding DAs are overloaded,
- Balance the load on all DAs by using a centralized RLB that uses a piecemeal iterative load-sensitive VD-DA weighted assignment algorithm to distribute the read I/O requests on each VD to least loaded DAs, thereby decreasing the probability of overloading a DA,
- Use a locality aware scheduler in each VD that chooses the optimal replica DA to submit a read I/O request, by adhering to both the locality in the input workload and the RLB weight distribution pattern,
- Use a hybrid flow control algorithm that uses a combination of QoS aware and QOS unaware techniques to regulate the flow of data between VDs and DAs.

In addition to the above mentioned research contributions, this work also demonstrates a solid implementation technique that overcomes the challenges in managing the centralized RLB algorithm and adopts the CFVC scheduler in a distributed storage setup. This work also convincingly demonstrates the correctness and effectiveness of the various proposed techniques using sophisticated simulations. Through comprehensive evaluations using both real-world

traces and synthetic workloads, we demonstrated the accuracy of PB extraction process and the effectiveness of centralized RLB technique.

Chapter 8

Conclusion and Future Directions

8.1 Conclusion

In this dissertation, we proposed several novel techniques that are applied across different components of a cloud storage system and in this section we summarize a few of our important research contributions.

Deduplication techniques for large-scale data workloads typically struggle to reduce CPU cycle utilization for disk block fingerprint comparisons and to avoid disk I/O bottlenecks in various components of the deduplication process. We built *Sungem* to use a novel sampling technique to design a highly useful and representative cache of the disk block fingerprints maintained in the main memory, that helps avoid disk I/O lookups for a large majority of the requests. Unlike typical deduplication techniques that store fingerprints occurring at the same time (temporal locality) in one place on the disk, *Sungem* uses spatial locality to store disk block fingerprints on disk and hence reduces both the number of fingerprint comparisons and the disk I/O lookups involved in prefetching the fingerprints to the main memory cache. *Sungem* also provides a large-scale storage block garbage collection solution, which to the best of our knowledge, is the first known garbage collection algorithm which is truly scalable in the sense that its bookkeeping overhead for each backup operation is proportional only to the size of the delta between consecutive backup operations. Using the above mentioned novel techniques, *Sungem* integrates deduplication and garbage collection solutions to process data at an extremely high speed of 7 TB/hour, using commodity hardware infrastructure, and this is atleast a 40% improvement over state-of-the-art sparse-indexing scheme [6] running with the same amount of hardware resources, for incremental backup

operations. We demonstrated through comprehensive evaluations that *Sungem* is able to deliver this high throughput consistently across different ranges of deduplication ratios, proving that *Sungem*'s performance is invariant to the number of duplicates in the data workload.

In the process of evaluating *Sungem*, we performed an in-depth characterization and analysis of a real-world trace that provides unique insights into the dynamics and caveats of modern deduplication algorithms in general. We applied the analysis to compare the relative merits and demerits of applying a bloom filter to disk data deduplication, and convincingly demonstrated that though a bloom filter helps in negatively filtering the unwanted disk I/O lookups in the process of locating duplicates for a disk block fingerprint, if its used in conjunction to an efficiently cached in-memory fingerprint index, then more often than not, the bloom filter hurts the deduplication systems performance. In the trace analysis, we showed the effectiveness of applying deduplication solution at various levels of granularities, including blocks of different sizes, and also demonstrated the advantages of empowering the deduplication process to be aware of the type of files which constitute the data workload.

Sungem mitigates the random disk access overhead in its garbage collection process using a novel disk access interface called BOSC, that supports disk update as a first-class primitive and enables the specification of application-specific callback functions to be invoked by the underlying storage system. Through comprehensive evaluations we showed that though the garbage collection process receives low locality workload to be processed by on-disk data structures, using BOSC, it completely amortizes the disk I/O activity using batched sequential sweeps on the hard disks. We also used the most widely used disk-based data structures, namely hash tables and B^+ trees, to demonstrate how easy it is to adopt BOSC interface and empirically demonstrated the efficiency of BOSC, to show that the update request throughput of a BOSC-based B^+ tree implementation is more than an order of magnitude faster than that of a vanilla B^+ tree built on top of the conventional disk access interface. Since B^+ trees and hash tables are widely used as disk-based data structures in several components of a cloud storage system, BOSC architecture can significantly improve the overall performance of the cloud storage system.

Since BOSC aggregates I/O requests in memory to improve its batching efficiency, we proposed to build *Beluga* to quickly log the data to a fast logging disk and ensure data persistency in case of a system crash. *Beluga*, is a high throughput, low latency, disk logging solution that delivers extremely high throughput for fine-grained disk I/O requests and hence provides a first-class performance similar to that of the popular but more expensive flash-based SSDs. *Beluga* fashions a carefully tuned disk write pipeline and makes it pos-

sible to provide on an array of three commodity 7200 RPM SATA disks, close to 5 million fine-grained (64-byte) disk logging operations per second, which is the maximum possible bandwidth on a commodity disk, while keeping the latency of each logging operation under 1 msec. Through *Beluga*, we convincingly demonstrated that flash-based SSDs are not the only means for a fast-logging disk solution. Since *Beluga* uses commodity SATA disks to deliver a performance as good as flash-based SSDs, we showed that the future of SATA disks still look very promising, especially given its cost-per-byte advantage and more importantly its capacity to read/write data for a long time without any short-range burn-out limits. We also proposed a energy saving version of *Beluga* that can dynamically adapt itself to consume lesser energy in proportion to the size of the incoming workload.

We carefully studied the most commonly faced difficulties of tenants of a cloud storage system and proposed *Cheetah* to build a QoS specification model that collects minimal information from the tenants about the specific performance expectations and more importantly, the specifications are collected completely in terms of what an application understands, unlike the existing approaches which forces the tenants to configure the QoS specifications using complicated system internal semantics like IOPS and MBPS. *Cheetah* enforces strict QoS guarantees using a combination of techniques like automatic load balancing using RLB that distributes the load in each VD to its replica DAs, while simultaneously preserving locality in the workload and also avoiding overloading of any DAs; flow control mechanism that regulates short-term bursts in data traffic between VDs and DAs, using a combination of QoS aware and QoS unaware flow control algorithms; QoS aware disk scheduler that ensures performance isolation between the VDs and simultaneously ensures optimal usage of hardware resources.

8.2 Future Directions

This dissertation addresses issues in several key components of a cloud storage system and hence we see several potential areas that deserve further research efforts in the line of our proposals made in this dissertation.

Sungem and most of the modern day deduplication solutions use fingerprint-based duplicate detection strategies, and hence there is a possibility of data loss because when two different data blocks falsely result in the same fingerprint due to hash collisions, one of them is garbage collected. Though the probability of such an event is extremely low, it is not affordable in many use-cases and hence an additional layer of duplicate check is employed to compare the entire data block byte-by-byte before declaring a block as a duplicate. Such

a byte-by-byte comparison process not only increases the CPU computations but also incurs disk bottleneck. It is interesting and challenging to tackle this issue because there is hardly any locality in the pair of fingerprints that match and hence the necessity to fetch in data blocks for further duplicate checks incur expensive random seeks to disk. Even if this additional layer of check is done lazily in the background, since any pair of potentially duplicate fingerprints have hardly any locality between them, each duplicate check incurs two random seeks on the disk and such a slow duplicate identification process doesn't scale well to cloud-scale storage systems.

Beluga demonstrates a state-of-the-art fast-logging technique that enables directly attached commodity SATA disks to be used for high throughput low latency disk logging applications. However, it is interesting to extend this idea to network attached disks, which could enable multiple user applications to use *Beluga* as a remote logging server that provides high throughput and low latency for fine-grained disk logging requests, without getting bothered by unpredictable network latencies. The main challenges are in ensuring micro second latency accuracy and in building a predictable latency model over the network.

In order to enable BOSC to handle large-scale random disk update workloads, one approach is adopt a distributed model in BOSC. For example, a distributed B^+ tree can split its internal nodes across several systems and BOSC architecture could be used internally within each such system in the distributed setup. An alternative approach is to use a centralized model, wherein the application generating random disk update requests could use a network attached storage server to satisfy large-scale storage requirements. In such a scenario, BOSC architecture could be applied to synchronously process data accesses across several disks in the remote storage server. However, in both the approaches, a challenging issue is to handle the callback function, since it has to be executed on a remote environment. A few challenging issues with remote execution of a callback function are to convincingly ensure safety of the user applications as well as that of the hardware resources from malicious user applications, to handle abnormal application behaviors at a remote location, to build a predictable latency model over the network, etc. Some of the safety-related solutions could be leveraged from the Active Disk [44] project, but it is interesting to convincingly resolve all these issues in a remotely attached storage environment.

Another interesting direction of research using BOSC is to adopt it in designing a generic file system. Few challenging issues are to handle multiple locks, synchronize data and metadata accesses to each disk I/O request and in providing security to callback functions in the context of a system call. It is

also interesting to handle read I/O workloads without disturbing the sequential sweep pattern in the background BOSC threads. Specifically, it is interesting to explore the relative merits and demerits of adopting a hierarchal storage setup using flash-based SSDs against the BOSC2 setup thats described in Section 5.2.4, or even to adopt a hybrid model that uses both these techniques.

BOSC and *Beluga* are heavily optimized for spinning disks, which are still among the favorites for disk-based storage devices. However, flash-based SSDs, shingled disks, phase change memory devices and a hybrid combination of these devices are forming an active area of research, of late. These newer technologies are fundamentally different in the way data is stored on disk and hence the core techniques employed in BOSC and *Beluga* do not work out-of-the-box with these newer technologies. Thankfully, manufacturers are increasingly opening up the control over these newer device internals, and its interesting to exploit such controls over the hardware to future-proof and extend the novel ideas used in BOSC and *Beluga*.

Cheetah enforces QoS guarantees at both VD and VDC level of granularity. However, when the VDs in the VDC do not share the same set of DAs, it is interesting to enforce QoS guarantee at VDC level granularity. One option is to coordinate between the DAs that collectively handle the workload from different VDs in the given VDC. But since such a coordination should happen in real-time, it is a challenging and interesting problem that deserves a worthwhile research effort. Another alternative is to group a set of DAs as a cluster and limit the amount of coordination between the DAs. However, it is not trivial to enforce such a constraint without wasting the storage hardware resources, given the highly volatile and complicated and shared disk access patterns across multiple VDs and DAs. Admission control and congestion control algorithms are also quite necessary to make *Cheetah* a viable commercial product, but both these algorithms involve extremely challenging and non-trivial issues.

Another very important direction of research work in *Cheetah* is to enforce latency guarantee. Since the latency of a disk I/O request involves both the network and disk latencies, it is extremely complicated and non-trivial to simultaneously build an accurate latency prediction model in a distributed storage setup and to avoid over provisioning of hardware resources just to cover up for the lack of techniques to bound the I/O latency of a disk access request.

We saw in Section 7.6.2 that hardware RAID controllers and checksum-based software RAID configurations pose severe challenges in adopting the CFVC scheduler. In order to ensure generic applicability of CFVC scheduler

across all types of DAs, it is necessary to fix the above mentioned incompatibility issue and thats another challenging research problem worthy of pursuing.

This dissertation focusses specifically on the block-level access interface of a cloud storage system, though it is not a limitation to extend the proposed ideas to other forms of access granularities like file-level, database-level and key-value stores. As an example, key-value stores, which are one of the very popular cloud storage system interfaces, can use *Sungem* and *Cheetah* with trivial modifications. The content proximity technique in *Sungem* and various characteristics of the data workload as analyzed in Chapter 4 suggest promising performance improvements when used in key-value stores.

8.3 Final Words

Together, with all the above mentioned research contributions, we propose to mitigate disk I/O bottlenecks in all the key components of a cloud storage system and with the deployment of *Cheetah*, we envision a cloud storage system which provides a perfectly virtualized disk access interface, providing 100% satisfaction to all workloads, irrespective of the locality and other implied constraints, while maximizing the utilization of hardware resources. Such a cloud storage system with 100% performance guarantees when combined with super fast internet speeds, can potentially handle real-time application's I/O workload, and that could open up a whole new research dimension into both the server-side and consumer electronics market. Though, by no means is this dissertation a one stop solution manual to all the issues in a cloud storage setup, we wish that our novel research contributions go a long way in building efficient cloud storage systems for the future.

Bibliography

[1] James Manyika, Michael Chui, Brad Brown, Jacques Bughin, Richard Dobbs, Charles Roxburgh, and Angela H Byers. Big data: The next frontier for innovation, competition, and productivity. 2011.

[2] John H Howard, Michael L Kazar, Sherri G Menees, David A Nichols, Mahadev Satyanarayanan, Robert N Sidebotham, and Michael J West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems (TOCS)*, 6(1):51–81, 1988.

[3] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1–10. IEEE, 2010.

[4] Raluca Ada Popa, Jacob R Lorch, David Molnar, Helen J Wang, and Li Zhuang. Enabling security in cloud storage slas with cloudproof. In *Proc. USENIX ATC*, 2011.

[5] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, et al. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.

[6] Mark Lillibridge, Kave Eshghi, Deepavali Bhagwat, Vinay Deolalikar, Greg Trezise, and Peter Camble. Sparse indexing: large scale, inline deduplication using sampling and locality. In *FAST '09: Proceedings of the 7th conference on File and storage technologies*, pages 111–123, 2009.

[7] Jeff Barr, Attila Narin, and Jinesh Varia. Building fault-tolerant applications on aws. *Amazon Web Services*, 2011.

[8] Mayur R Palankar, Adriana Iamnitchi, Matei Ripeanu, and Simson Garfinkel. Amazon s3 for science grids: a viable solution? In *Pro-*

ceedings of the 2008 international workshop on Data-aware distributed computing, pages 55–64. ACM, 2008.

[9] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 307–320, Berkeley, CA, USA, 2006. USENIX Association. ISBN 1-931971-47-1. URL http://dl.acm.org/citation.cfm?id=1298455.1298485.

[10] 0. An introduction to gluster architecture. https://confluence.oceanobservatories.org/download/attachments/30998760/An_Introduction_To_Gluster_ArchitectureV7_110708.pdf, .

[11] Kazutaka Morita. Sheepdog: Distributed storage system for qemu/kvm. *LCA 2010 DS&R miniconf*, 2010.

[12] RJ Honicky and Ethan L Miller. Replication under scalable hashing: A family of algorithms for scalable decentralized data distribution. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, page 96. IEEE, 2004.

[13] Tom White. *Hadoop: The Definitive Guide: The Definitive Guide*. O'Reilly Media, 2009.

[14] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[15] EMC. *EMC Isilon OneFS: A Technical Overview*. EMC, 2013.

[16] EMC. Hadoop on emc isilon scale-out nas. http://www.emc.com/collateral/software/white-papers/h10528-wp-hadoop-on-isilon.pdf, 2012.

[17] Jean-Pierre Le Goaller, Carlos Conde, and Shakil Langha. Rdbms in the cloud: Oracle database on aws. 2013.

[18] Mocky Habeeb. *A Developer's Guide to Amazon SimpleDB*. Addison-Wesley Professional, 2010.

[19] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly

available key-value store. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 205–220. ACM, 2007.

[20] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, Jaidev Haridas, Chakravarthy Uddaraju, Hemal Khatri, Andrew Edwards, Vaman Bedekar, Shane Mainali, Rafay Abbasi, Arpit Agarwal, Mian Fahim ul Haq, Muhammad Ikram ul Haq, Deepali Bhardwaj, Sowmya Dayanand, Anitha Adusumilli, Marvin McNett, Sriram Sankaran, Kavitha Manivannan, and Leonidas Rigas. Windows azure storage: A highly available cloud storage service with strong consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 143–157, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0977-6. doi: 10.1145/2043556.2043571. URL http://doi.acm.org/10.1145/2043556.2043571.

[21] 0. Solidfire solution overview. http://bit.ly/OSaZoG, .

[22] 0. Pure storage flash array. http://www.purestorage.com/pdf/Pure_Datasheet_FA400.pdf, .

[23] 0. Ibm storwize ovewview. http://www-03.ibm.com/systems/storage/disk/storwize_v7000/overview.html, .

[24] 0. Vmware vsphere ovewview. http://www.vmware.com/products/datacenter-virtualization/vsphere/overview.html, .

[25] Laura M. Grupp, John D. Davis, and Steven Swanson. The bleak future of nand flash memory. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, FAST'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association. URL http://dl.acm.org/citation.cfm?id=2208461.2208463.

[26] Eduardo Pinheiro, Wolf-Dietrich Weber, and Luiz André Barroso. Failure trends in a large disk drive population. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies*, FAST '07, pages 2–2, Berkeley, CA, USA, 2007. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1267903.1267905.

[27] Gokul Soundararajan, Vijayan Prabhakaran, Mahesh Balakrishnan, and Ted Wobber. Extending ssd lifetimes with disk-based write caches. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies*, FAST'10, pages 8–8, Berkeley, CA, USA, 2010. USENIX

Association. URL http://dl.acm.org/citation.cfm?id=1855511.1855519.

[28] David Sacks. Demystifying storage networking das, san, nas, nas gateways, fibre channel, and iscsi. *IBM Storage Networking*, pages 3–11, 2001.

[29] 0. http://www.opencompute.org/wp/wp-content/uploads/2012/05/Open-Vault-Storage-Specification-v0.5.pdf, .

[30] Chien-Yung Lee, Yu-Wei Lee, Cheng-Chun Tu, Pai-Wei Wang, Yu-Cheng Wang, Chih-Yu Lin, and Tzi cker Chiueh. Autonomic failover for a software-defined container computer network. In *Presented as part of the 10th International Conference on Autonomic Computing*, pages 225–234, Berkeley, CA, 2013. USENIX. ISBN 978-1-931971-02-7. URL https://www.usenix.org/conference/icac13/technical-sessions/presentation/lee.

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

[32] Cezary Dubnicki, Leszek Gryz, Lukasz Heldt, Michal Kaczmarczyk, Wojciech Kilian, Przemyslaw Strzelczak, Jerzy Szczepkowski, Cristian Ungureanu, and Michal Welnicki. Hydrastor: a scalable secondary storage. In *FAST '09: Proceedings of the 7th conference on File and storage technologies*, pages 197–210, 2009.

[33] EMC Corp. EMC Centera: Content Addressed Storage System. http://www.emc.com/collateral/hardware/data-sheet/c931-emc-centera-cas-ds.pdf, 2008.

[34] Sean Rhea, Russ Cox, and Alex Pesterev. Fast, inexpensive content-addressed storage in foundation. In *ATC'08: USENIX 2008 Annual Technical Conference on Annual Technical Conference*, pages 143–156, 2008.

[35] PUB FIPS. 180-1. secure hash standard. *National Institute of Standards and Technology*, 17, 1995.

[36] Deepavali Bhagwat, Kave Eshghi, Darrell D. E. Long, and Mark Lillibridge. Extreme binning: Scalable, parallel deduplication for chunk-based

file backup. In *MASCOTS'09: Proceedings of the the 17th IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, september 2009.

[37] Benjamin Zhu, Kai Li, and Hugo Patterson. Avoiding the disk bottleneck in the data domain deduplication file system. In *FAST'08: Proceedings of the 6th USENIX Conference on File and Storage Technologies*, pages 1–14, 2008.

[38] Lawrence L You, Kristal T Pollack, and Darrell DE Long. Deep store: An archival storage system architecture. In *Data Engineering, 2005. ICDE 2005. Proceedings. 21st International Conference on*, pages 804–815. IEEE, 2005.

[39] Exagrid. Comparing exagrids byte-level data de-duplication to block level data de-duplication. http://www.bmrturkey.com/downloads/exagrid/Data%20De-duplication%20Methodologies.pdf, 2010.

[40] Biplob Debnath, Sudipta Sengupta, and Jin Li. Chunkstash: speeding up inline storage deduplication using flash memory. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, pages 16–16. USENIX Association, 2010.

[41] Austin Clements, Irfan Ahmad, Murali Vilayannur, and Jinyuan Li. Decentralized deduplication in san cluster file systems. In *ATC'09: 2009 USENIX Annual Technical Conference*, pages 101–114, 2009.

[42] Atish Kathpal, Matthew John, and Gaurav Makkar. Distributed duplicate detection in post-process data de-duplication. HiPC, 2011.

[43] William J. Bolosky, Scott Corbin, David Goebel, and John R. Douceur. Single instance storage in windows 2000. In *WSS'00: Proceedings of the 4th conference on USENIX Windows Systems Symposium*, pages 2–2, 2000.

[44] NetApp. Open systems snapvault (ossv) best practices guide. https://communities.netapp.com/servlet/JiveServlet/previewBody/4791-102-2-13466/tr-3466.pdf.

[45] Andrew Tridgell, Paul Mackerras, et al. The rsync algorithm, 1996.

[46] Michael O Rabin. *Fingerprinting by random polynomials*. Center for Research in Computing Techn., Aiken Computation Laboratory, Univ., 1981.

[47] Athicha Muthitacharoen, Benjie Chen, and David Mazières. A low-bandwidth network file system. *SIGOPS Oper. Syst. Rev.*, 35(5):174–187, October 2001. ISSN 0163-5980. doi: 10.1145/502059.502052. URL http://doi.acm.org/10.1145/502059.502052.

[48] Erik Kruus, Cristian Ungureanu, and Cezary Dubnicki. Bimodal content defined chunking for backup streams. In *FAST*, pages 239–252, 2010.

[49] BTony Asaro and Heidi Biggar. Data de-duplication and disk-to-disk backup systems: Technical and business considerations. *The Enterprise Strategy Group*, 2007.

[50] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/362686.362692.

[51] Petros Efstathopoulos and Fanglu Guo. Rethinking deduplication scalability. *HotStorage 2010*, 2010.

[52] P. Armangau and S.R. Dunham. Computer data storage backup with tape overflow control of disk caching of backup data stream, April 15 2003. URL http://www.google.com/patents/US6549992. US Patent 6,549,992.

[53] Gartner. http://www.mainframezone.com/storage/backup-recovery-business-continuity/tape-a-collapsing-star.

[54] Dan Feng, Lingfang Zeng, Fang Wang, and Peng Xia. Tlfs: High performance tape library file system for data backup and archive. In *Proceedings of 7th International Meeting on High Performance Computing for Computational Science. Rio de Janeiro, Brazil: Springer*, 2006.

[55] John R. Douceur, Atul Adya, William J. Bolosky, Dan Simon, and Marvin Theimer. Reclaiming space from duplicate files in a serverless distributed file system. In *ICDCS '02: Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS'02)*, page 617, 2002. ISBN 0-7695-1585-1.

[56] Dirk Meister and André Brinkmann. Multi-level comparison of data deduplication in a backup scenario. In *SYSTOR '09: Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, pages 1–12, 2009. ISBN 978-1-60558-623-6. doi: http://doi.acm.org/10.1145/1534530.1534541.

[57] George Forman, Kave Eshghi, and Jaap Suermondt. Efficient detection of large-scale redundancy in enterprise file systems. *ACM SIGOPS Operating Systems Review*, 43(1):84–91, 2009. ISSN 0163-5980. doi: http://doi.acm.org/10.1145/1496909.1496926.

[58] Partho Nath, Bhuvan Urgaonkar, and Anand Sivasubramaniam. Evaluating the usefulness of content addressable storage for high-performance data intensive applications. In *HPDC '08: Proceedings of the 17th international symposium on High performance distributed computing*, pages 35–44, 2008. ISBN 978-1-59593-997-5. doi: http://doi.acm.org/10.1145/1383422.1383428.

[59] Nitin Agrawal, William J. Bolosky, John R. Douceur, and Jacob R. Lorch. A five-year study of file-system metadata. *Trans. Storage*, 3(3), October 2007. ISSN 1553-3077. doi: 10.1145/1288783.1288788. URL http://doi.acm.org/10.1145/1288783.1288788.

[60] John R. Douceur and William J. Bolosky. A large-scale study of file-system contents. In *Proceedings of the 1999 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, SIGMETRICS '99, pages 59–70, New York, NY, USA, 1999. ACM. ISBN 1-58113-083-X. doi: 10.1145/301453.301480. URL http://doi.acm.org/10.1145/301453.301480.

[61] Kylie M. Evans and Geoffrey H. Kuenning. A study of irregularities in file-size distributions. In *In International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS 02*, 2002.

[62] Andrew W. Leung, Shankar Pasupathy, Garth Goodson, and Ethan L. Miller. Measurement and analysis of large-scale network file system workloads. In *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, ATC'08, pages 213–226, Berkeley, CA, USA, 2008. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1404014.1404030.

[63] Daniel Ellard, Jonathan Ledlie, Pia Malkani, and Margo Seltzer. Passive nfs tracing of email and research workloads. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, FAST '03, pages 203–216, Berkeley, CA, USA, 2003. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1090694.1090716.

[64] Ahmed El-Shimi, Ran Kalach, Ankit Kumar, Adi Oltean, Jin Li, and Sudipta Sengupta. Primary data deduplication-large scale study and system design. In *Proceedings of the 2012 USENIX conference on Annual Technical Conference*, USENIX ATC'12, pages 26–26, Berkeley, CA, USA, 2012. USENIX Association. URL http://dl.acm.org/citation.cfm?id=2342821.2342847.

[65] Nohhyun Park and D.J. Lilja. Characterizing datasets for data deduplication in backup applications. In *Workload Characterization (IISWC), 2010 IEEE International Symposium on*, pages 1–10, 2010. doi: 10.1109/IISWC.2010.5650369.

[66] Grant Wallace, Fred Douglis, Hangwei Qian, Philip Shilane, Stephen Smaldone, Mark Chamness, and Windsor Hsu. Characteristics of backup workloads in production systems. In *Proceedings of the 10th USENIX conference on File and Storage Technologies*, FAST'12, pages 4–4, Berkeley, CA, USA, 2012. USENIX Association. URL http://dl.acm.org/citation.cfm?id=2208461.2208465.

[67] Dutch T. Meyer and William J. Bolosky. A study of practical deduplication. *Trans. Storage*, 7(4):14:1–14:20, February 2012. ISSN 1553-3077. doi: 10.1145/2078861.2078864. URL http://doi.acm.org/10.1145/2078861.2078864.

[68] Dominique Colnet, Philippe Coucaud, and Olivier Zendra. Compiler support to customize the mark and sweep algorithm. In *ISMM '98: Proceedings of the 1st international symposium on Memory management*, pages 154–165, 1998. ISBN 1-58113-114-3. doi: http://doi.acm.org/10.1145/286860.286877.

[69] Fanglu Guo and Petros Efstathopoulos. Building a high-performance deduplication system. In *USENIXATC'11: Proceedings of the 2011 USENIX Conference on USENIX annual technical conference*, 2011.

[70] Rifat Shahriyar, Stephen Michael Blackburn, Xi Yang, and Kathryn S McKinley. Taking off the gloves with reference counting immix. In *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*, pages 93–110. ACM, 2013.

[71] George E Collins. A method for overlapping and erasure of lists. *Communications of the ACM*, 3(12):655–657, 1960.

[72] David F Bacon, Clement R Attanasio, VT Rajan, Stephen E Smith, and HANB LEE. A pure reference counting garbage collector.

[73] Andreas Haeberlen, Alan Mislove, and Peter Druschel. Glacier: Highly durable, decentralized storage despite massive correlated failures. In *Proceedings of the 2Nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2*, NSDI'05, pages 143–158, Berkeley, CA, USA, 2005. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1251203.1251214.

[74] Dilip Nijagal Simha, Maohua Lu, and Tzi-cker Chiueh. An update-aware storage system for low-locality update-intensive workloads. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '12, pages 375–386, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-0759-8. doi: 10.1145/2150976.2151016. URL http://doi.acm.org/10.1145/2150976.2151016.

[75] Transaction Processing Performance Council. *TPC Benchmark C Standard Specification*, volume 1 and 2. Waterside Associates, Fremont, CA, 1.0.a edition, Aug, 1996.

[76] Cyril U. Orji and Jon A. Solworth. Write-Only Disk Cache Experiments on Multiple Surface Disks. In *ICCI '92: Proceedings of the Fourth International Conference on Computing and Information*, pages 385–388, Washington, DC, USA, 1992. IEEE Computer Society. ISBN 0-8186-2812-X.

[77] Wenguang Wang, Yanping Zhao, and Rick Bunt. HyLog: A High Performance Approach to Managing Disk Layout. In *FAST '04: Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, pages 145–158, Berkeley, CA, USA, 2004. USENIX Association.

[78] Mendel Rosenblum and John K. Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems (TOCS)*, 10(1):26–52, 1992. ISSN 0734-2071. doi: http://doi.acm.org/10.1145/146941.146943.

[79] Gregory R. Ganger, Marshall Kirk McKusick, Craig A. N. Soules, and Yale N. Patt. Soft Updates: a Solution to the Metadata Update Problem in File Systems. *ACM Transactions on Computer Systems (TOCS)*, 18 (2):127–153, 2000. ISSN 0734-2071. doi: http://doi.acm.org/10.1145/350853.350863.

[80] Marshall Kirk McKusick and Gregory R. Ganger. Soft Updates: a Technique for Eliminating Most Synchronous Writes in the Fast Filesystem. In *ATEC'99: Proceedings of the Annual Technical Conference on 1999 USENIX Annual Technical Conference*, pages 24–24, Berkeley, CA, USA, 1999. USENIX Association.

[81] S. Chutani, O. T. Anderson, M. L. Kazar, B. W. Leverett, W. A. Mason, and R. N. Sidebotham. The Episode File System. In *Proceedings of the USENIX Winter 1992 Technical Conference*, pages 43–60, San Fransisco, CA, USA, 1992. USENIX Association.

[82] R. Hagmann. Reimplementing the Cedar File System using Logging and Group Commit. In *SOSP '87: Proceedings of the 11th ACM Symposium on Operating Systems Principles*, pages 155–162, New York, NY, USA, 1987. ACM Press. ISBN 0-89791-242-X. doi: http://doi.acm.org/10.1145/41457.37518.

[83] Michael Stonebraker. The Design of the POSTGRES Storage System. In *VLDB '87: Proceedings of the 13th International Conference on Very Large Data Bases*, pages 289–300, San Francisco, CA, USA, 1987. Morgan Kaufmann Publishers Inc. ISBN 0-934613-46-X.

[84] C. Chao, R. English, D. Jacobson, A. Stepanov, and J. Wilkes. Mime: A High-Performance Parallel Storage Device with Strong Recovery Guarantees. Technical Report HPL-CSP-92-9 rev 1, HewlettPackard Laboratories Report, November 1992.

[85] Donald D. Chamberlin, Morton M. Astrahan, Michael W. Blasgen, James N. Gray, W. Frank King, Bruce G. Lindsay, Raymond Lorie, James W. Mehl, Thomas G. Price, Franco Putzolu, Patricia Griffiths Selinger, Mario Schkolnick, Donald R. Slutz, Irving L. Traiger, Bradford W. Wade, and Robert A. Yost. A History and Evaluation of System R. *Communications of the ACM*, 24(10):632–646, 1981. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/358769.358784.

[86] Soumyadeb Mitra, Windsor W. Hsu, and Marianne Winslett. Trustworthy Keyword Search for Regulatory-Compliant Records Retention. In *VLDB '2006: Proceedings of the 32nd International Conference on Very Large Data Bases*, pages 1001–1012. VLDB Endowment, 2006.

[87] Berthier Ribeiro-Neto, Edleno S. Moura, Marden S. Neubert, and Nivio Ziviani. Efficient Distributed Algorithms to Build Inverted Files. In

SIGIR '99: Proceedings of the 22nd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, pages 105–112, New York, NY, USA, 1999. ACM Press. ISBN 1-58113-096-1. doi: http://doi.acm.org/10.1145/312624.312663.

[88] Jun Hirai, Sriram Raghavan, Hector Garcia-Molina, and Andreas Paepcke. WebBase: a Repository of Web Pages. In *Proceedings of the 9th International World Wide Web Conference on Computer Networks : the International Journal of Computer and Telecommunications Networking*, pages 277–293, Amsterdam, The Netherlands, 2000. North-Holland Publishing Co. doi: http://dx.doi.org/10.1016/S1389-1286(00)00063-3.

[89] Sergey Melnik, Sriram Raghavan, Beverly Yang, and Hector Garcia-Molina. Building a Distributed Full-Text Index for the Web. In *WWW '01: Proceedings of the 10th International Conference on World Wide Web*, pages 396–406, New York, NY, USA, 2001. ACM Press. ISBN 1-58113-348-0. doi: http://doi.acm.org/10.1145/371920.372095.

[90] Goetz Graefe. B-tree Indexes for High Update Rates. *SIGMOD Rec.*, 35(1):39–44, 2006. ISSN 0163-5808. doi: http://doi.acm.org/10.1145/1121995.1122002.

[91] Laurynas Biveinis, Simonas Šaltenis, and Christian S. Jensen. Main-memory operation buffering for efficient R-tree update. In *VLDB 2007: Proceedings of the 33rd International Conference on Very Large Data Bases*, pages 591–602. VLDB Endowment, 2007. ISBN 978-1-59593-649-3.

[92] Michael A. Bender, Gerth Stolting Brodal, Rolf Fagerberg, Dongdong Ge, Simai He, Haodong Hu, John Iacono, and Alejandro Lopez-Ortiz. The cost of cache-oblivious searching. In *Proceedings of FOCS 2003*, pages p. 271–282, 2003.

[93] Michael A. Bender, Erik D. Demaine, and Martin Farach-Colton. Cache-oblivious b-trees. In *SIAM Journal of Computing*, volume 35(2), pages p. 341–358, 2005.

[94] Michael A. Bender, Jeremy T. Fineman, Seth Gilbert, and Bradley C. Kuszmaul. Concurrent cache-oblivious b-trees. In *Proceedings of SPAA 2005*, pages p. 228–237, 2005.

[95] Lars Arge, Klaus Hinrichs, Jan Vahrenhold, and Jeffrey Scott Vitter. Efficient bulk operations on dynamic r-trees. *Algorithmica*, 33(1):104–128, 2002. URL citeseer.ist.psu.edu/arge99efficient.html.

[96] Lars Arge. The buffer tree: A new technique for optimal i/o-algorithms (extended abstract). In *WADS '95: Proceedings of the 4th International Workshop on Algorithms and Data Structures*, pages 334–345, London, UK, 1995. Springer-Verlag. ISBN 3-540-60220-8.

[97] O. Procopiuc, P. Agarwal, L. Arge, and J. Vitter. Bkd-tree: A dynamic scalable kd-tree, 2002. URL citeseer.ist.psu.edu/procopiuc02bkdtree.html.

[98] Goetz Graefe. Write-optimized B-trees. In *VLDB 2004: Proceedings of the Thirtieth International Conference on Very Large Data Bases*, pages 672–683. VLDB Endowment, 2004. ISBN 0-12-088469-0.

[99] Edmund B. Nightingale, Kaushik Veeraraghavan, Peter M. Chen, and Jason Flinn. Rethink the Sync. In *OSDI'2006: Proceedings of the 7th conference on USENIX Symposium on Operating Systems Design and Implementation*, pages 1–14, Berkeley, CA, USA, 2006. USENIX Association.

[100] Maxim Lifantsev and Tzi-cker Chiueh. I/O-Conscious Data Preparation for Large-Scale Web Search Engines. In *VLDB '2002: Proceedings of the 32nd International Conference on Very Large Data Bases*, pages 382–393, Hongkong, China, 2002. VLDB Endowment.

[101] Taher Haveliwala. Efficient Computation of PageRank. Technical Report 1999-31, Stanford University, Stanford University, Feburary 1999. URL citeseer.ist.psu.edu/haveliwala99efficient.html.

[102] Yen-Yu Chen, Qingqing Gan, and Torsten Suel. I/O-Efficient Techniques for Computing PageRank. In *CIKM '02: Proceedings of the 11th International Conference on Information and Knowledge Management*, pages 549–557, New York, NY, USA, 2002. ACM Press. ISBN 1-58113-492-4. doi: http://doi.acm.org/10.1145/584792.584882.

[103] Raymie Stata, Krishna Bharat, and Farzin Maghoul. The Term Vector Database: Fast Access to Indexing Terms for Web Pages. In *Proceedings of the 9th International World Wide Web Conference on Computer Networks : the International Journal of Computer and Telecommunications Networking*, pages 247–255, Amsterdam, The Netherlands, 2000. North-Holland Publishing Co. doi: http://dx.doi.org/10.1016/S1389-1286(00)00046-3.

[104] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The PageRank Citation Ranking: Bringing Order to the Web. Technical Report 1999-66, Stanford Digital Library Technologies Project, 1998. URL citeseer.ist.psu.edu/page98pagerank.html.

[105] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. Aries: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst.*, 17(1):94–162, March 1992. ISSN 0362-5915. doi: 10.1145/128765.128770. URL http://doi.acm.org/10.1145/128765.128770.

[106] Ross S. Finlayson and David R. Cheriton. Log files: An extended file service exploiting write-once storage. In *SOSP '87 Proceedings of the eleventh ACM Symposium on Operating systems principles*, NY, USA, 1987. ISBN 0-89791-242-X.

[107] Tzi-cker Chiueh. Trail: a track-based logging disk architecture for zero-overhead writes. In *Computer Design: VLSI in Computers and Processors, 1993. ICCD '93. Proceedings., 1993 IEEE International Conference on*, pages 339 – 343, 1993.

[108] Jimmy P. Strickland, Peter P. Uhrowczik, and Vern L. Watts. Ims/vs: An evolving system. In *IBM Systems Journal Volume 21*, pages 490 – 510, 1982.

[109] HAGMANN R. Low latency logging. http://www.bitsavers.org/pdf/xerox/parc/techReports/CSL-91-1_Low_Latency_Logging.pdf, 1991.

[110] Klaus Elhardt and Rudolf Bayer. A database cache for high performance and fast restart in database systems. In *ACM Transactions on Database Systems (TODS), Volume 9 Issue 4*, pages 503–525, 1984.

[111] Tzi-cker Chiueh and Lan Huang. Track-based disk logging. In *in Proceedings of International Conference on Dependable Systems and Networks*, pages 429–438, 2002.

[112] Christopher R. Lumb, Jiri Schindler, and Gregory R. Ganger. Freeblock scheduling outside of disk firmware. In *Proceedings of the 1st USENIX conference on File and storage technologies*, FAST'02, pages 20–20, Berkeley, CA, USA, 2002. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1973333.1973353.

[113] Bill Gallagher, Dean Jacobs, and Anno Langen. A high-performance, transactional filestore for application servers. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, SIGMOD '05, pages 868–872, New York, NY, USA, 2005. ACM. ISBN 1-59593-060-4. doi: 10.1145/1066157.1066269. URL http://doi.acm.org/10.1145/1066157.1066269.

[114] Jongmin Gim and Youjip Won. Extract and infer quickly: Obtaining sector geometry of modern hard disk drives. In *ACM Transactions on Storage (TOS), Volume 6 Issue 2*, NY, USA, 2010.

[115] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers, Third Edition*. O'Reilly Media, 2005.

[116] Ying Chen, Windsor W. Hsu, and Honesty C. Young. Logging raid an approach to fast, reliable, and low-cost disk arrays. In *Proceeding Euro-Par '00 Proceedings from the 6th International Euro-Par Conference on Parallel Processing*, pages 1302–1312, 2000.

[117] Dushyanth Narayanan, Austin Donnelly, Eno Thereska, Sameh Elnikety, and Antony Rowstron. Everest: Scaling down peak loads through i/o off-loading. In *In Proceedings of OSDI*, pages 15–28, 2008.

[118] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. In *Journal ACM Transactions on Computer Systems (TOCS),Volume 10 Issue 1*, pages 26–52, 1992.

[119] Hui Dai, Michael Neufeld, and Richard Han. Elf: an efficient log-structured flash file system for micro sensor nodes. In *SenSys '04 Proceedings of the 2nd international conference on Embedded networked sensor systems*, 2004.

[120] Y Hu and Q Yang. Dcd - disk caching disk: A new approach for boosting i/o performance. In *Proceedings of the 23rd International Symposium on Computer Architecture*, pages 169–178, 1996.

[121] A. Kawaguchi, S. Nishioka, and H. Motoda. A flash-memory based file system. In *USENIX Winter*, pages 155–164, 1995.

[122] M. Wu and W. Zwaenepoel. envy: a non−volatile, main memory storage system. In *ASPLOS,*, pages 86–97, 1994.

[123] F. Douglis, R. Caceres, M. Frans Kaashoek, K. Li, B. Marsh, and J. A. Tauber. Storage alternatives for mobile computers. In *In Proceedings of*

the First Symposium on Operating Design and Implementation (OSDI), 1994.

[124] Sang-Won Lee and Bongki Moon. Design of flash-based dbms: An in-page logging approach. In *In Proceedings of the ACM SIGMOD*, pages 55–66, Beijing, China, 2007.

[125] Shimin Chen. Flashlogging: exploiting flash devices for synchronous logging performance. In *SIGMOD '09 Proceedings of the 35th SIGMOD international conference on Management of data*, NY, USA, 2009. ACM. ISBN 978-1-60558-551-2.

[126] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Architecting phase change memory as a scalable dram alternative. In *in Proceedings of ISCA09*, pages 2–13, 2009.

[127] Ru Fang, Hui-I Hsiao, Bin He, C. Mohan, and Yun Wang. High performance database logging using storage class memory. In *IEEE 27th International Conference on Data Engineering*, pages 1221 – 1231, Hannover, Germany, 2011. icde. ISBN 978-1-4244-8959-6.

[128] R. Freitas and W. Wilcke. Storage-class memory: The next storage system technology. In *IBM Journal of Research and Development, Vol. 52, Issue 4*, pages 439 – 447, 2008.

[129] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *SOSP '07 Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, NY, USA, 2007. ACM. ISBN 978-1-59593-591-5.

[130] Avinash Lakshman and Prashant Malik. Cassandra - a decentralized structured storage system. http://www.cs.cornell.edu/projects/ladis2009/papers/lakshman-ladis2009.pdf, 2009.

[131] John A. Chandy. A dual actuator logging disk architecture. In *Journal of Systems Architecture: the EUROMICRO Journal, Volume 53 Issue 12*, pages 913–926, 2007.

[132] Peter M Chen. Optimizing delay in delayed-write file systems. In *In Proceedings of the 1994 International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 1994.

[133] John Wilkes. Traveling to rome: Qos specifications for automated storage system management. In *Quality of ServiceIWQoS 2001*, pages 75–91. Springer, 2001.

[134] Douglas B. Terry, Vijayan Prabhakaran, Ramakrishna Kotla, Mahesh Balakrishnan, Marcos K. Aguilera, and Hussam Abu-Libdeh. Consistency-based service level agreements for cloud storage. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 309–324, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2388-8. doi: 10.1145/2517349.2522731. URL http://doi.acm.org/10.1145/2517349.2522731.

[135] Peng Gang and Tzi-cker Chiueh. Availability and fairness support for storage qos guarantee. In *Distributed Computing Systems, 2008. ICDCS '08. The 28th International Conference on*, pages 589–596, 2008. doi: 10.1109/ICDCS.2008.107.

[136] Lan Huang, Gang Peng, and Tzi-cker Chiueh. Multi-dimensional storage virtualization. *SIGMETRICS Perform. Eval. Rev.*, 32(1):14–24, June 2004. ISSN 0163-5999. doi: 10.1145/1012888.1005692. URL http://doi.acm.org/10.1145/1012888.1005692.

[137] Lan Huang. Stonehenge: A high performance virtualized storage cluster with qos guarantees. 2003.

[138] Ming Zhao and Renato J. Figueiredo. Experimental study of virtual machine migration in support of reservation of cluster resources. In *Proceedings of the 2Nd International Workshop on Virtualization Technology in Distributed Computing*, VTDC '07, pages 5:1–5:8, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-897-8. doi: 10.1145/1408654.1408659. URL http://doi.acm.org/10.1145/1408654.1408659.

[139] Ripal Nathuji, Aman Kansal, and Alireza Ghaffarkhah. Q-clouds: managing performance interference effects for qos-aware clouds. In *Proceedings of the 5th European conference on Computer systems*, pages 237–250. ACM, 2010.

[140] Ajay Gulati, Ganesha Shanmuganathan, Xuechen Zhang, and Peter Varman. Demand based hierarchical qos using storage resource pools. In *Proceedings of the 2012 USENIX conference on Annual Technical Conference*, USENIX ATC'12, pages 1–1, Berkeley, CA, USA, 2012. USENIX Association. URL http://dl.acm.org/citation.cfm?id=2342821.2342822.

[141] VMware. Storage drs: Automated management of storage devices in a virtualized datacenter. http://labs.vmware.com/academic/publications/drs-vmtj-winter2012, 2012.

[142] David Shue, Michael J. Freedman, and Anees Shaikh. Performance isolation and fairness for multi-tenant cloud storage. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 349–362, Berkeley, CA, USA, 2012. USENIX Association. ISBN 978-1-931971-96-6. URL http://dl.acm.org/citation.cfm?id=2387880.2387914.

[143] Ajay Gulati, Irfan Ahmad, Carl A Waldspurger, et al. Parda: Proportional allocation of resources for distributed storage access. In *FAST*, volume 9, pages 85–98, 2009.

[144] Ajay Gulati, Chethan Kumar, Irfan Ahmad, and Karan Kumar. Basil: automated io load balancing across storage devices. In *Proceedings of the 8th USENIX conference on File and storage technologies*, FAST'10, pages 13–13, Berkeley, CA, USA, 2010. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1855511.1855524.

[145] Matthew Wachs, Michael Abd-El-Malek, Eno Thereska, and Gregory R. Ganger. Argon: performance insulation for shared storage servers. In *Proceedings of the 5th USENIX conference on File and Storage Technologies*, FAST '07, pages 5–5, Berkeley, CA, USA, 2007. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1267903.1267908.

[146] Anna Povzner, Tim Kaldewey, Scott Brandt, Richard Golding, Theodore M. Wong, and Carlos Maltzahn. Efficient guaranteed disk request scheduling with fahrrad. *SIGOPS Oper. Syst. Rev.*, 42(4):13–25, April 2008. ISSN 0163-5980. doi: 10.1145/1357010.1352595. URL http://doi.acm.org/10.1145/1357010.1352595.

[147] Christopher R. Lumb, Arif Merchant, and Guillermo A. Alvarez. Façade: Virtual storage devices with performance guarantees. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, FAST '03, pages 131–144, Berkeley, CA, USA, 2003. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1090694.1090710.

[148] D. Ferrari and D.C. Verma. A scheme for real-time channel establishment in wide-area networks. *Selected Areas in Communications, IEEE Journal on*, 8(3):368–379, 1990. ISSN 0733-8716. doi: 10.1109/49.53013.

[149] M. Shreedhar and George Varghese. Efficient fair queueing using deficit round robin. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '95, pages 231–242, New York, NY, USA, 1995. ACM. ISBN 0-89791-711-1. doi: 10.1145/217382.217453. URL http://doi.acm.org/10.1145/217382.217453.

[150] Magnus Karlsson, Christos Karamanolis, and Xiaoyun Zhu. Triage: Performance differentiation for storage systems using adaptive control. *Trans. Storage*, 1(4):457–480, November 2005. ISSN 1553-3077. doi: 10.1145/1111609.1111612. URL http://doi.acm.org/10.1145/1111609.1111612.

[151] Theodore M Wong, Richard A Golding, Caixue Lin, and Ralph A Becker-Szendy. Zygaria: Storage performance as a managed resource. In *Real-Time and Embedded Technology and Applications Symposium, 2006. Proceedings of the 12th IEEE*, pages 125–134. IEEE, 2006.

[152] Ajay Gulati, Arif Merchant, and Peter J Varman. pclock: an arrival curve based approach for qos guarantees in shared storage systems. *ACM SIGMETRICS Performance Evaluation Review*, 35(1):13–24, 2007.

[153] Jianyong Zhang, Anand Sivasubramaniam, Qian Wang, Alma Riska, and Erik Riedel. Storage performance virtualization via throughput and latency control. *Trans. Storage*, 2(3):283–308, August 2006. ISSN 1553-3077. doi: 10.1145/1168910.1168913. URL http://doi.acm.org/10.1145/1168910.1168913.

[154] Guillermo A. Alvarez, Elizabeth Borowsky, Susie Go, Theodore H. Romer, Ralph Becker-Szendy, Richard Golding, Arif Merchant, Mirjana Spasojevic, Alistair Veitch, and John Wilkes. Minerva: An automated resource provisioning tool for large-scale storage systems. *ACM Trans. Comput. Syst.*, 19(4):483–518, November 2001. ISSN 0734-2071. doi: 10.1145/502912.502915. URL http://doi.acm.org/10.1145/502912.502915.

[155] Eric Anderson, Michael Hobbs, Kimberly Keeton, Susan Spence, Mustafa Uysal, and Alistair Veitch. Hippodrome: Running circles around storage administration. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, FAST '02, Berkeley, CA, USA, 2002. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1083323.1083341.

[156] Chenyang Lu, Guillermo A Alvarez, and John Wilkes. Aqueduct: Online data migration with performance guarantees. In *FAST*, volume 2, page 21. Citeseer, 2002.

[157] John Wilkes, Richard Golding, Carl Staelin, and Tim Sullivan. The hp autoraid hierarchical storage system. *ACM Transactions on Computer Systems (TOCS)*, 14(1):108–136, 1996.

[158] Christopher Stewart, Aniket Chakrabarti, and Rean Griffith. Zoolander: Efficiently meeting very strict, low-latency slos. In *Proceedings of the 10th International Conference on Autonomic Computing (ICAC 13)*, pages 265–277, San Jose, CA, 2013. USENIX. ISBN 978-1-931971-02-7. URL https://www.usenix.org/conference/icac13/technical-sessions/presentation/stewart.

[159] Christina Delimitrou and Christos Kozyrakis. Paragon: Qos-aware scheduling for heterogeneous datacenters. In *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS '13, pages 77–88, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1870-9. doi: 10.1145/2451116.2451125. URL http://doi.acm.org/10.1145/2451116.2451125.

[160] Thomas E. Anderson, Susan S. Owicki, James B. Saxe, and Charles P. Thacker. High-speed switch scheduling for local-area networks. *ACM Trans. Comput. Syst.*, 11(4):319–352, November 1993. ISSN 0734-2071. doi: 10.1145/161541.161736. URL http://doi.acm.org/10.1145/161541.161736.

[161] Nick McKeown. The islip scheduling algorithm for input-queued switches. *IEEE/ACM Trans. Netw.*, 7(2):188–201, April 1999. ISSN 1063-6692. doi: 10.1109/90.769767. URL http://dx.doi.org/10.1109/90.769767.

[162] M. Katevenis, S. Sidiropoulos, and C. Courcoubetis. Weighted round-robin cell multiplexing in a general-purpose atm switch chip. *Selected Areas in Communications, IEEE Journal on*, 9(8):1265–1279, 1991. ISSN 0733-8716. doi: 10.1109/49.105173.

[163] Kartik Gopalan, Lan Huang, Gang Peng, Tzi-Cker Chiueh, and Yow-Jian Lin. Statistical admission control using delay distribution measurements. *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMCCAP)*, 2(4):258–281, 2006.

[164] L. Dubois and R. Amatruda. IDC Backup and Recovery/Data Deduplication report. Technical Report, IDC and EMC, Feb 2010.

[165] Fanglu Guo and Tzi-cker Chiueh. DAFT: Disk Geometry-Aware File System Traversal. In *MASCOTS'09: Proceedings of the the 17th IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 56–68, 2009.

[166] Dilip Nijagal Simha, Maohua Lu, and Tzi-cker Chiueh. A scalable deduplication and garbage collection engine for incremental backup. In *Proceedings of the 6th International Systems and Storage Conference*, SYSTOR '13, pages 16:1–16:12, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2116-7. doi: 10.1145/2485732.2485753. URL http://doi.acm.org/10.1145/2485732.2485753.

[167] Jiansheng Wei, Hong Jiang, Ke Zhou, Dan Feng, and Hua Wang. Detecting duplicates over sliding windows with ram-efficient detached counting bloom filter arrays. In *NAS: 6th IEEE International Conference on Networking, Architecture, and Storage*, 2011.

[168] Guanlin Lu, Biplob Debnath, and David H.C. Du. A forest-structured bloom filter with flash memory. In *MSST: Storage Conference*, 2011.

[169] Biplob Debnath, Sudipta Sengupta, Jin Li, David J. Lilja, and David H.C. Du. Bloomflash: Bloom filter on flash-based storage. In *ICDCS: International Conference on Distributed Computing Systems*, 2011.

[170] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z. Broder. Summary cache: A scalable wide-area web cache sharing protocol. *IEEE/ACM Trans. Netw.*, 8(3):281–293, June 2000. ISSN 1063-6692. doi: 10.1109/90.851975. URL http://dx.doi.org/10.1109/90.851975.

[171] Sasu Tarkoma, Christian Esteve Rothenberg, and Eemil Lagerspetz. Theory and practice of bloom filters for distributed systems. *Communications Surveys and Tutorials, IEEE*, 99:1 – 25, 2011. ISSN 1553-877X. doi: 10.1109/SURV.2011.031611.00024.

[172] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, July 1970. ISSN 0001-0782. doi: 10.1145/362686.362692. URL http://doi.acm.org/10.1145/362686.362692.

[173] Changwoo Min, Kangnyeon Kim, Hyunjin Cho, Sang-Won Lee, and Young Ik Eom. Sfs: Random write considered harmful in solid state drives. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, FAST'12, pages 12–12, Berkeley, CA, USA, 2012. USENIX Association. URL http://dl.acm.org/citation.cfm?id=2208461.2208473.

[174] Zhifeng Chen, Yan Zhang, Yuanyuan Zhou, Heidi Scott, and Berni Schiefer. Empirical Evaluation of Multi-level Buffer Cache Collaboration for Storage Systems. *SIGMETRICS Perform. Eval. Rev.*, 33(1):145–156, 2005. ISSN 0163-5999. doi: http://doi.acm.org/10.1145/1071690.1064230.

[175] Lakshmi N. Bairavasundaram, Muthian Sivathanu, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. X-RAY: A Non-Invasive Exclusive Caching Mechanism for RAIDs. In *ISCA '04: Proceedings of the 31st annual international symposium on Computer architecture*, page 176, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2143-6.

[176] Asit Dan and Don Towsley. An Approximate Analysis of the LRU and FIFO Buffer Replacement Schemes. *SIGMETRICS Perform. Eval. Rev.*, 18(1):143–152, 1990. ISSN 0163-5999. doi: http://doi.acm.org/10.1145/98460.98525.

[177] Norman P. Jouppi. Cache Write Policies and Performance. In *ISCA '93: Proceedings of the 20th annual international symposium on Computer architecture*, pages 191–201, New York, NY, USA, 1993. ACM Press. ISBN 0-8186-3810-9. doi: http://doi.acm.org/10.1145/165123.165154.

[178] Li Ou Xubin Ben He, Martha J. Kosa, and Stephen L. Scott. A Unified Multiple-Level Cache for High Performance Storage Systems. In *MASCOTS '05: Proceedings of the 13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 143–152, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2458-3. doi: http://dx.doi.org/10.1109/MASCOT.2005.10.

[179] Christopher R. Lumb, Jiri Schindler, and Gregory R. Ganger. Freeblock Scheduling Outside of Disk Firmware. In *FAST '02: Proceedings of the Conference on File and Storage Technologies*, pages 275–288, Berkeley, CA, USA, 2002. USENIX Association. ISBN 1-880446-03-0.

[180] Chi Zhang, Xiang Yu, Arvind Krishnamurthy, and Randolph Y. Wang. Configuring and Scheduling an Eager-Writing Disk Array for a Transaction Processing Workload. In *FAST '02: Proceedings of the 1st USENIX Conference on File and Storage Technologies*, page 24, Berkeley, CA, USA, 2002. USENIX Association.

[181] Chris Malakapalli and Vamsi Gunturu. Evaluation of SCSI over TCP/IP and SCSI over Fibre Channel Connections. In *HOTI '01: Proceedings of the The Ninth Symposium on High Performance Interconnects (HOTI '01)*, page 87, Washington, DC, USA, 2001. IEEE Computer Society. ISBN 0-7695-1357-3.

[182] Gordon F. Hughes and Joseph F. Murray. Reliability and Security of RAID Storage Systems and D2D Archives using SATA Disk Drives. *Transactions on Storage*, 1(1):95–107, 2005. ISSN 1553-3077. doi: http://doi.acm.org/10.1145/1044956.1044961.

[183] Ananth Devulapalli, Dennis Dalessandro, Pete Wyckoff, and Nawab Ali. Attribute Storage Design for Object-based Storage Devices. In *MSST '07: Proceedings of the 24th IEEE Conference on Mass Storage Systems and Technologies*, pages 263–268, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-3025-7. doi: http://dx.doi.org/10.1109/MSST.2007.4.

[184] V.; Yongdae Kim Kher. Decentralized Authentication Mechanisms for Object-based Storage Devices. In *SISW '03: Proceedings of the Second IEEE International Security in Storage Workshop*, page 1, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-2059-6.

[185] InnoDB. *The InnoDB Storage Engine*. URL http://dev.mysql.com/doc/refman/5.5/en/innodb-storage-engine.html.

[186] Dilip Nijagal Simha, Tzi-cker Chiueh, Ganesh Karuppur Rajagopalan, and Pallav Bose. High-throughput low-latency fine-grained disk logging. In *Proceedings of the ACM SIGMETRICS/international conference on Measurement and modeling of computer systems*, SIGMETRICS '13, pages 255–266, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1900-3. doi: 10.1145/2465529.2465552. URL http://doi.acm.org/10.1145/2465529.2465552.

[187] Erik Riedel, Christos Faloutsos, Garth A. Gibson, and David Nagle. Active disks: Remote execution for network-attached storage. Technical Report CMU-CS-97-198, Parallel Data Lab, Carnegie Mellon University,

December 1997. URL "http://www.pdl.cmu.edu/PDL-FTP/Active/activedisks01.pdf".

[188] Maohua Lu, Shibiao Lin, and Tzi-cker Chiueh. Efficient Logging and Replication Techniques for Comprehensive Data Protection. In *MSST '07: Proceedings of the 24th IEEE Conference on Mass Storage Systems and Technologies*, pages 171–184, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-3025-7. doi: http://dx.doi.org/10.1109/MSST.2007.14.

[189] Darren Erik Vengroff and Jeffrey Scott Vitter. I/O-Efficient Algorithms and Environments. *ACM Computing Surveys (CSUR)*, 28(4):212, 1996. ISSN 0360-0300. doi: http://doi.acm.org/10.1145/242224.242495.

[190] Lars Arge, Octavian Procopiuc, and Jeffrey Scott Vitter. Implementing I/O-Efficient Data Structures Using TPIE. In *ESA '02: Proceedings of the 10th Annual European Symposium on Algorithms*, pages 88–100, London, UK, 2002. Springer-Verlag. ISBN 3-540-44180-8.

[191] Open Source Development Labs (OSDL). Database Test Suite: DBT-[1,2,3,4,5]. http://osdldbt.sourceforge.net/, 2003.

[192] Anthony Hylick, Ripduman Sohan, Andrew Rice, and Brian Jones. An analysis of hard drive energy consumption. In *Proceedings of 16th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS 2008)*, pages 103 – 112, 2008.

[193] Seagate and Intel. Serial ata native command queuing. www.seagate.com/content/pdf/whitepaper/D2c_tech_paper_intc-stx_sata_ncq.pdf, 2003.

[194] E. Coffman, L. Klimko, and B. Ryan. Analysis of scanning policies for reducing disk seek times. *SIAM Journal on Computing*, 1(3):269–279, 1972. doi: 10.1137/0201018. URL http://epubs.siam.org/doi/abs/10.1137/0201018.

[195] Margo Seltzer, Peter Chen, and John Ousterhout. Disk scheduling revisited. In *In Proceedings of the USENIX Winter Technical Conference (USENIX Winter 90*, pages 313–324, 1990.

[196] J. Nagle. On packet switches with infinite storage. *Communications, IEEE Transactions on*, 35(4):435–438, 1987. ISSN 0090-6778. doi: 10.1109/TCOM.1987.1096782.

[197] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. *SIGCOMM Comput. Commun. Rev.*, 19(4): 1–12, August 1989. ISSN 0146-4833. doi: 10.1145/75247.75248. URL http://doi.acm.org/10.1145/75247.75248.

[198] L. Zhang. Virtual clock: a new traffic control algorithm for packet switching networks. In *Proceedings of the ACM symposium on Communications architectures & protocols*, SIGCOMM '90, pages 19–29, New York, NY, USA, 1990. ACM. ISBN 0-89791-405-8. doi: 10.1145/99508.99525. URL http://doi.acm.org/10.1145/99508.99525.

[199] John S Bucy, Jiri Schindler, Steven W Schlosser, and Gregory R Ganger. The disksim simulation environment version 4.0 reference manual (cmu-pdl-08-101). *Parallel Data Laboratory*, page 26, 2008.

[200] Vasily Tarasov, Gyumin Sim, Anna Povzner, and Erez Zadok. Efficient i/o scheduling with accurately estimated disk drive latencies. *OSPERT*, 2012:36, 2012.

[201] John Zedlewski, Sumeet Sobti, Nitin Garg, Fengzhou Zheng, Arvind Krishnamurthy, Randolph Y Wang, et al. Modeling hard-disk power consumption. In *FAST*, volume 3, pages 217–230, 2003.

[202] Jiri Schindler Gregory and Gregory R Ganger. Automated disk drive characterization. 1999.

[203] Ken Bates and Bruce McNutt. Umass storage trace repository. http://traces.cs.umass.edu/index.php/Storage/Storage, 2007.